Report No. UIUCDCS-R-74-662 NSF - OCA - GJ-328 - 000002

PREPAGING AND APPLICATIONS TO STRUCTURED ARRAY PROBLEMS

by

Kishor Shridharbhai Trivedi

July 1974

DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-74-662

PREPAGING AND APPLICATIONS TO STRUCTURED ARRAY PROBLEMS*

by

Kishor Shridharbhai Trivedi

July 1974

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by the National Science Foundation under Grant No. US NSF GJ-328 and was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, July 1974.

ACKNOWLEDGMENT

The author wishes to express his gratitude to Professor J. Richard Phillips for his guidance during the preparation of this thesis.

I would like to thank the National Science Foundation and the Department of Computer Science for financial support. Thanks are also due to Mrs. Vivian Alsip and Mrs. June Wingler for typing this thesis.

Finally, I would like to thank my wife, Kalpana, for her encouragement and patience.

TABLE OF CONTENTS

Page

LIST OF FIGURES

1. INTRODUCTION

Paged virtual memory systems (PVMS) were introduced by Kilburn et al [KILB62] in the Ferranti Atlas computer. The objective of PVMS is to relieve the programmer of the burden of storage management. However, the cost, in terms of overhead, and performance degradation was, at the beginning, thought to be high [FINE66,KUEH68]. Nevertheless, others argued in favor of PVMS [DENN70,SAYR69], and PVMS are well accepted and they have proved to be useful in practice [DENN70]. But the effectiveness and the overhead leave much to be desired [MASU72,ORGA72]. This is especially true in terms of the changing technology with very large memories and the types of problems with large data bases and large arrays.

In this thesis, we are concerned with the performance of programs running under a PVMS and with techniques and algorithms that include prepaging as a more viable solution for the computer systems of the future.

In Chapter 2, we introduce the terminology used and survey the relevant literature. We identify variables of a PVMS which affect the performance of its programs. We discuss the question of what performance measures to use. We then study the effect of each of the variables of PVMS on the performance, from the available literature. We focus our attention on two important variables of a PVMS, namely, the paging algorithm used by the system and the locality of the programs. We note that the most popular paging algorithms are demand paging algorithms

chiefly because of their simplicity in implementation and because little is known about prepaging. The property of locality has been observed in practical programs and is, perhaps, the chief reason of the practicability of PVMS [DENN70]. For special applications, people have given methods to write "more local" programs. Compiler implemented locality improvement methods have restricted them to code and not data. Further, these methods only do the physical reorganization of code and not a logical reorganization.

In Chapter 3, we present several new paging algorithms, some of them being prepaging algorithms. We show why prepaging is useful. We present a new algorithm, DPMIN, which is a demand-prepaging algorithm, and prove that it is an optimal demand-prepaging algorithm. However, DPMIN cannot be implemented in practice since it requires that the program's reference string be known in advance. We also present several practical prepaging algorithms. We then present a variable-memory prepaging algorithm called PWS, which is based on Denning's WS algorithm and prove that it incurs zero page faults. PWS algorithm is also impractical in the same sense as DPMIN and is only useful for theoretical purposes. We also study the question of performance measurement while using these prepaging algorithms, in particular, we study whether or not the proposed paging algorithms are stack algorithms [MATT70].

In Chapter 4, we show how to improve the locality of matrix algorithms. This includes a logical reorganization of the program and its application to several common matrix algorithms. We also measure and compare the average working set size of programs and show that this is a reasonable way to measure locality.

In Chapter 5, we introduce prepaging in matrix algorithms. By using the prepaging algorithms proposed in Chapter 3, we show how to improve the performance of the matrix algorithms.

In Chapter 6, we discuss the automation of the performance-improvement techniques of Chapters 4 and 5. Prepaging applied to matrix algorithms can be implemented in a compiler but not in an operating system. Eventually one would hope that these techniques would find their way into some of the compilers of the future and thereby provide efficient prepaging that cannot be obtained in other ways.

2. GENERAL PAGING

2.0 Introduction

In a paged virtual memory system (PVMS), a program's address space is divided into fixed-size blocks of addresses called pages and the main memory (MM) is divided into matching size blocks of locations called page frames.

We will let $N = \{1, 2, \ldots, n\}$ denote a program's address space and $M = \{1, 2, \ldots, c\}$ denote the set of MM page frames alloted to the program, and generally $1 \leq c \leq n$. When a program is executing, it makes a sequence of references to its address space N. We will denote such a sequence by $\omega = r_1, r_2, \ldots, r_t, \ldots r_T$, where $r_t \in N$, $t \geq 1$. We will let $|\omega|$ denote the number of references in ω, N^0 denote the set containing the null string Λ, $N^T = \{\omega = r_1, r_2, \ldots, r_t, \ldots \mid |\omega| = T\}$, N^* denote the set of all finite strings on N and $N^+ = N^* - \{\Lambda\}$.

Now since the CPU (central processing unit) can only refer to the MM, we must interpose a mapping mechanism between the CPU and the MM. The address map f, then, is a function, $f: N \rightarrow M$. Since it is possible that $c < n$, f may be a partial function as follows:

$$f(i) = \begin{cases} j & \text{if page } i \text{ resides in page frame } j \\ & \text{undefined otherwise} \end{cases}$$

The function f is also dynamic, since in general, a program may refer to any of the pages of N. If the page size is p, a valid virtual address is an integer α, $0 \leq \alpha < np$. Similarly, a valid MM address is an integer β, $0 \leq \beta < cp$. When presented with the virtual address α, the address translation mechanism (MAP) obtains (i, ℓ) such that $\alpha = (i-1)\ p + \ell$ where $0 \leq \ell < p$, and then generates memory address $\beta = [f(i)-1]\ p + \ell$ if $f(i)$ is defined and a 'page fault' if $f(i)$ is undefined. A page fault interrupts the execution of the program and the program makes a transition from running state to the page-wait state. If we assume that the whole of the program's address space resides on a secondary memory device such as a disc, a drum or a bulk core store, then upon the occurrence of a page fault, the memory manager has to bring the required page from secondary memory to MM. The program waits for the required page for the duration of one page fetch time, assumed to be, on the average, T time units. If the MM was originally full, the above process will also force us to choose a page, already resident in MM, to be replaced. The process of page replacement may also involve a page push from MM to the secondary memory if it was written into during its last stay in MM. After the required page has been brought into MM, the interrupted program is in the ready state, ready to execute on a CPU if one is available. During the time of the above page transfer, the CPU will remain idle unless we resort to multiprogramming. In multiprogramming, we try to mask a long page transfer time by overlapping I/O and execution. After a page fault interrupt, the CPU is alloted to another ready program if one is available. If the system does not allow multi-programming, then it is a monoprogrammed system.

It has been observed in practice that a page fault incurs a significant amount of overhead. As an example, Masuda [MASU72] reports that in his system, more than 50% of running time of the operating system is spent servicing page fault interrupts and 30% of the time is a pure idle time during which the cpu has no task to execute and is waiting for the interrupt of the I/O termination. The situation is not much different in other PVMS [ORGA72]. Thus it seems that, it is very important to study the performance of a PVMS and find methods to improve performance. Improving performance of a program running under PVMS will obviously improve the performance of the PVMS. We will investigate the performance of programs running under a PVMS.

2.1 Performance Measures

To study the performance, we need a set of measures. Furthermore, performance as seen by the user and as seen by the system are two different ways to study this topic. From the point of view of system efficiency, the number of page faults π, the number of page pulls C, and space time product ST, of MM used are three important performance measures that we will consider. From the user point of view, the cost of running the program and the turnaround time are important performance measures. In a monoprogrammed system, user oriented measures are directly related to the number of page faults, thus in such a system it will be sufficient to study the page fault measure. In a multiprogrammed system, user oriented measures cannot be so easily related to the system oriented measures. In most systems page faults or page pulls do not occur explicitly in the cost charged to the user. However, a few milliseconds

of cpu time per page fault is generally required. Since the cost is generally determined by the cpu time used and memory usage, we see that the cost is affected by page faults and the ST product. In a similar way, the turnaround time is affected, at least indirectly, by the number of page faults incurred. Because of these reasons, we will consider only the system oriented performance measures.

A page fault implies the following sequence of operations: interruption of the program and a switch to the page fault handler, search for the required page on the secondary memory device, search for a page to be replaced, issuing of output instructions to push the page to be replaced (if required), issuing of input instructions to fetch the required page, searching the ready queue and finally a process switch to a new program from the ready queue. All these operations take considerable amount of time, in fact, a few milliseconds per page fault [MASU72, ORGA72]. Thus if we want to improve the cpu utilization, we must minimize the number of page faults. Another effect of a page fault is poor memory utilization. For instance, during the page wait time and also during the time the process spends in the ready queue waiting for a cpu, the page frames of MM alloted to the program are unavailable to other processes.

The number of page pulls has a direct relationship to the channel traffic. With a given system, the maximum amount of channel traffic that it can support is fixed. This implies that the page pulls incurred by an individual program is an important measure of its behavior. In multi-programmed systems there is an extreme situation, called thrashing, in which excessive page traffic occurs [DENN68]. Thrashing has the effect of rapidly decreasing cpu utilization, and it is therefore important to

eliminate it if possible. Note that, if we use demand paging, then the page fault and the page pull measures become indistinguishable. A related performance measure is the number of page pushes; however, it is a fixed percentage of the number of page pulls, on the average.

During the page-wait time, the set of pages alloted to the interrupted program is wasted. If these pages were alloted to the currently executing program, we could improve its performance. Therefore, the utilization of MM can be measured by the space time product of MM used by a program during the entire time of its residence in MM.

2.2 Important Variables

There are seven important variables in a PVMS which can affect the performance. These are:

1. Whether the system is monoprogrammed or multiprogrammed.
2. The number of allocated page frames c.
3. The page size p.
4. The average page fetch time T, and the ratio T/Δ where Δ is the memory cycle time.
5. The memory referencing behavior of the program, which is reflected in the page reference string ω.
6. The organization of the program's information (both instructions and data) in the virtual address space.
7. The paging algorithm used.

There are four controllers of these seven variables: hardware, operating system, compiler and the programmer [WEIN72].

The hardware controls the page size p, the average page fetch time T, the memory cycle time Δ, and the total amount of MM available. In some systems it also controls the paging algorithm used [LIPT68].

The operating system controls the degree of multiprogramming used, the number of allocated page frames c and whether or not c is allowed to vary dynamically, and finally, it determines the paging algorithm used.

The compiler decides the page reference string, at least partially, insofar as the translation process allows the freedom to do so. Storage allocation phase of the compiler organizes a program's information in the virtual address space.

The programmer may control the page reference string and to a certain extent, the organization of the program's information in the virtual address space.

The effect of these seven variables on the performance will be considered now in more detail.

2.2.1 Multiprogramming

We have observed that multiprogramming is used to overlap the page transfers for one program with the execution of another program, and as a result it has a potential of improving the cpu utilization. We note that multiprogramming also has other uses, such as, to provide time sharing, but in this work we concern ourselves with the former use only. The number of programs occupying the MM concurrently is known as the degree of multiprogramming. The basic questions in such a system are what degree of multiprogramming to use and how much improvement in cpu utilization is accrued as a result. If a very low degree is used,

we may not be able to find a ready job to run on the occurrence of a page fault; as a result the cpu will remain idle. On the other hand, whenever several concurrent programs share MM in order to 'mask' I/O time each program gets a smaller number of MM page frames which tends to increase page faulting. In an excellent survey article, Kuck, et al. [KUCK70] discuss this question in great depth. Their conclusion is that for multiprogramming to yield a reasonable gain in cpu utilization, there must be 'sufficient' amount of MM. They have also shown that with other variables being fixed, the cpu utilization, generally, goes up starting from a degree of multiprogramming equal to one (i.e., monoprogramming) until it reaches a maximum and then it falls down very quickly. If the degree of multiprogramming is very high, the system is susceptible to thrashing, a collapse of performance that may occur when MM is overcommitted [DENN68, DENN70]. Denning has proposed that by using a working set scheduling strategy, thrashing can be avoided. We will not discuss the question of degree of multiprogramming any further. Interested readers are referred to [KUCK70, DENN70, COFF73, SMIT67].

2.2.2 Main Memory Allotment

The effect of MM allotment on page faults has been studied in a number of papers [BELA69, DENN68, DENN68a, KUCK70, LEHM68, OPPE68, SHEM66, SHEM69, SISS68, SMIT67, FINE66, MATT70, SALT74], and many experiments have been conducted to determine program paging behavior [ANAC67, BAER71, BAYL68, BELA66, BRAW68, COFF68, DENN70, FINE66, FREI68, JOSE70, LEHM68, ONEI67, PINK68, SALT74, STEV68, VARI68]. Conclusion of all these studies seems to be that the paging rate is very high at low

page allotments and it reduces very fast as the allotment is increased. After an amount of page allotment called 'parachor' [KUEH68] is reached, the decrease in paging rate achieved by increasing c is not very significant. The value of c at parachor is dependent on the particular program under consideration. This immediately brings us to the point of fixed versus variable size page allotment.

In a monoprogrammed system, page allotment is fixed and such a question does not arise; however, if we have a multiprogrammed system then control could be exercised. It has been shown that variable memory allotment is generally superior to fixed memory allotment [BELA69, COFF72, DENN72].

2.2.3 The Page Size

The effect of page size p on the performance of a PVMS has been extensively studied in the literature [BAER71, DENN70, DOYL , GELE71, GELE73, HATF72, JOSE70, KUCK70, RAND68, RAND69]. There are several factors which affect the choice of a page size. Operating system typically rounds the number of pages up to an integral number of pages which results in unused words called internal fragmentation [RAND69]. Reducing the page size will reduce internal fragmentation. Kuck [KUCK70] defines superfluity to be the unreferenced words on a page during the interval of residence of the page in MM. Doyle [DOYL] has called this gap fragmentation. Reducing the page size tends to minimize this problem. By decreasing the size of the page, more space must be kept in the page table which leaves less MM available for program paging. This effect is called table fragmentation [DENN70, DOYL]. A balance should be achieved

to minimize these fragmentation problems. Note that fragmentation only affects the memory utilization. There are other effects of the page size. With a fixed MM size, reducing page size seems to have beneficial effect on the page faults [ANAC67, BAER71, BAYL68, COFF68, JOSE70]. Decreasing the page size, however, increases cpu overhead, the hardware cost of the address translation mechanism, the channel traffic and average word fetch time [DENN70]. We will not study the question of page size in this work.

2.2.4 The Page-Fetch Time

The effect of the average page fetch time T on the performance of a PVMS is the next point to be discussed. In a monoprogrammed system, the cpu is idle during this time, so a large T implies a very low cpu utilization and also a large turnaround time. In a multiprogramming system, the effect of a large T is masked by overlapping page-wait time of one program with the execution of another. But, as we have seen, multiprogramming implies memory sharing, which in turn implies more page faulting. If we avoid memory sharing and resort to swapping then the channel traffic is increased.

Kuck [KUCK70] has concluded that for $T > 6000$ units, there is no gain to be had from multiprogramming. Here T is measured in average instruction time units (roughly microseconds). Roughly speaking, this means that movable head disc is a very bad paging device. Drums and fixed head disc fall below the 6000 mark, and are useful. Extended core storage (ECS) provides $T < 1000$ and is an excellent paging device. In fact, with ECS, it has been recommended that the process switch after a page fault be avoided altogether and the cpu be left idle during the page-wait time [ORGA72].

Denning [DENN68] reports that the main reason for thrashing can be traced to a large value of T. For a further discussion see [DENN70].

2.2.5 Program Behavior

The memory referencing behavior of a program is a very important variable in determining performance of a PVMS. A program which scatters its references over the entire address space will page fault heavily. It is therefore desirable that the program tend to refer to a small subset of its address space in a small time interval. Fortunately, the above property, known as the locality property, is found to be exhibited by most programs in practice [BELA66, COFF73, DENN68, DENN70, DENN72a, FINE66, LIPT68, BRAW68, BRAW70, MCKE69, KUEH68, SAYR69, SHED72]. In fact, this property of locality is the key reason of the feasibility of PVMS.

More specifically, the property of locality can be summarized in three statements [DENN72a]:

> During any time interval, a program distributes its references nonuniformly over its address space, some pages being favored over the others.
>
> The density of references to a given page changes slowly in time or the set of favored pages changes membership slowly.
>
> Two disjoint segments of the page reference string tend to be highly correlated when the interval between them is short, and tend to become uncorrelated as the interval between them increases.

It is clear that the behavior and style of the programmer has a direct bearing on the locality of programs. Programmers tend to use sequential and looping control structures and they tend to concentrate on small sections of large programs for moderately long time intervals, and they generally group data into content-related blocks.

The set of favored pages seem naturally to split into four areas of activity [GIBS66, JOSE70]. One of these areas is constituted by the instruction addresses of the program. The other three areas are constituted by data addresses. The processes of data analysis often consist of arithmetic or logical manipulations on two or more strings of independently (semi) coherent addresses which in combination form the sequence of operand addresses. The fourth area of activity is the output catchment area where the results are held prior to outputting. If it is not possible to study the program behavior in this great a detail, it may be profitable to separate the set of favored pages into data pages and instruction pages [DENN72]. We will investigate this topic in detail in this thesis. We remark that for problems with large data bases, the instruction paging is almost trivial compared with the data paging. Therefore, we will concentrate, for the most part, on the data paging problem in this thesis.

Examples of what may happen to performance of a program with poor locality are too numerous [FINE66, BRAW68, DENN65, BAIR68, SMIT67]. Therefore, it is very important that methods be developed to create guidelines on how to write programs with a high degree of locality. Before we discuss these locality improvement methods, we discuss several other results on program behavior.

To study and predict performance of programs running under PVMS, several models of program behavior have been investigated. These models are used to generate reference strings to be used in the analytical study of program behavior. The validation of a proposed model is another question that need be answered. The simplest of all models investigated

is the independent reference model, which has been studied by several authors [COFF73, KING71, FRAN74, AHO71, BELA66]. In this model, it is assumed that the probability of a reference to page i at time t is given by, $P_r[r_t = i] = \beta_i \quad \forall\, t \geq 1 \quad \forall\, 1 \leq i \leq n$, where the set of all β_i is fixed and $\sum_{i=1}^{n} \beta_i = 1$. It has been observed that, this model is relatively simple to analyze but it does not mirror the behavior of actual programs [COFF73]. Aho et al [AHO71] proposed a very general ℓ-order nonstationary model of program behavior. However, except in 0-order stationary case (which is identical to the independent reference model), very few results are obtained for this general model. Denning et al [DENN72a] propose several locality models of program behavior. They were able to produce some interesting results with these models which compared favorably with the behavior of practical programs. Mattson et al [MATT70] propose an LRU stack model of program behavior which was improved and modified by Shedler et al [SHED72]. In the LRU stack model, it is assumed that the locality L_t of the program at time t is of a fixed size c, equal to the page allotment, and consists of the last c pages referred by the program. Quite clearly, this is a good model. Saltzer [SALT74] has recently proposed a simple linear model of program behavior which is validated against the MULTICS system. This model assumes that the page fault probability is inversely proportional to the page allotment c.

Several authors have given guidelines to produce programs with better locality [BRAW68, DENN70, FINE66, HATF71, KUEH68, WEIN72]. The use of modular and highly structured programming and programming languages is a suggestion made most often. The improvement of locality by proper

arrangement of code and data in VA space is a problem we will discuss separately. Here we are concerned with logical organization of the program. When asked to solve a particular problem, a programmer, generally, has a choice of a variety of algorithms to use. As an example, if he is to solve a system of linear equations, he can use Gaussian elimination, LU decomposition or Cholesky factorization and many others [ISAA66]. The choice of the algorithm is based upon the mathematical properties of data; therefore, we expect the locality to vary with each algorithm. Even when a particular algorithm is chosen, it may be possible to change the order of some operations without affecting the algorithm drastically. We would like to consider the order of operations which yields the maximum locality. In order to discuss this topic in detail, we have to consider individual applications. For applications involving sorting, see [BRAW70], for searching problem see [KNIG], and for the list processing application see [BOBR67]. In the case of matrix algorithms, McKeller et al [MCKE69] have shown that block algorithms have a superior locality property compared with nonblock algorithms. Dubrulle [DUBR72] has discussed the solution of the Eigenvalue problem in a paged environment. Moler [MOLE72] has discussed a method of loop reversal for improving matrix algorithms. Rogers [ROGE73] discusses the solution of linear equations in a paged environment. We will discuss methods of improving locality of matrix algorithms. Our work will be distinguished from that of the above by the fact that we will be studying many more matrix algorithms and that our measurements will be more extensive. In particular, we will study matrix multiplication, Cholesky decomposition, LU decomposition, Gaussian

elimination and Gram Schmidt orthonormalization. We note here that, we choose to ignore the error growth aspect of our algorithms unlike Rogers. We will measure the paging performance of our algorithms under a variety of paging algorithms, both realizable and unrealizable. As against this, McKeller et al have only measured the performance of their algorithms under MIN [BELA66] paging algorithm, which is unrealizable.

So far we have discussed only programmer-implemented locality improvement methods. One of the motivations for adopting a virtual memory system is that it relieves the programmer from the burden of memory management (or overlay problems) when his program cannot fit into the available MM [DENN70, KILB62], therefore, asking the programmer to worry about locality of his programs seems to be a step backwards. The job of improving locality of programs must then be carried out by the computer system. The operating system cannot possibly do much, since it does not know the global structure of the program. Therefore, it appears plausible that the optimization we seek can be achieved by the compiler. We know of no earlier work along these lines. Observe that, we are considering the topic of logical program reorganization and not physical reorganization. We will consider this type of compiler optimization for matrix algorithms written in both a high-level language (FORTRAN or PL/I) and an array language OL/2 [PHIL72]. We remark that the OL/2 language is an array language which allows dynamic partitioning and block structure, but it does not allow a 'GOTO' statement. These characteristics greatly influence the program reorganization strategies.

2.2.6 Pagination

The organization of a program's information in the VA space has been termed pagination [DENN70]. This can be further divided into code (instruction) pagination and data pagination. In code pagination, at the highest level one could study the grouping of subroutines to minimize page faults. Such experiments were made by Comeau [COME67]. Code pagination can be done on the basis of statistical properties (e.g., branch probabilities) [RAMA66, DENN70], the history collected from previous runs [HATF71, FERR73], or solely on the basis of syntax [YELO71]. The problem of data pagination has received very little attention. In particular, only for special applications like matrix algorithms have investigations been made. It has been noted by several authors [MCKE69, ROGE73] that storing matrices by square blocks coupled with the use of block algorithms seems to improve paging performance. We will assume block (or submatrix) type of storage for matrices.

2.2.7 Paging Algorithms

We now discuss paging algorithms and their effect on paging performance. A paging algorithm implements three policies. The fetch policy determines when a particular page should be brought into MM; a replacement policy determines when a particular page is removed from MM; and a placement policy determines an available page frame to hold a fetched page. In a PVMS (as against a segmented VMS), the placement policy is trivial and therefore, will not be considered [DENN70].

Basically there are two types of fetch policies, namely, demand fetching and anticipatory fetching. Under demand fetching, a page is

brought into MM only on demand, i.e., when a page is referenced and is not found to be resident in MM (i.e., at the time of a page fault). Under anticipatory fetch policy (also called prefetching), one or more pages may be brought into MM at any time, usually in advance of reference to the page. A special kind of prefetching policy is called demand prefetching [COFF73]. Assume that at time t, a page x is referenced and is not found in MM, thus creating a page fault. As a result a page fetch for page x will be initiated. A demand prefetch policy will allow us to fetch some other pages together with page x in the above situation.

A replacement policy can also be one of two types: demand an anticipatory.

Most popular paging algorithms are of demand_fetch, demand_replace variety and are commonly known as demand paging algorithms. All the other varieties of paging algorithms are commonly known as prepaging algorithms. Demand paging has been widely used in practice due to its simplicity in implementation. Prepaging, on the other hand, is more difficult to implement, chiefly because good predictions of a program's pages needed in future are not easily obtained [DENN70, DENN72].

Two types of prepaging schemes have been reported. One scheme, known as swapping, has been used to guard against excessive paging due to a small time quantum (e.g., time-sharing systems) [DENN70, KUEH68, ORGA72]. In this scheme (except for the first time quantum) a program's working set is preloaded at the beginning of the time quantum and demand paging is used during the time quantum. We note that the working set to be preloaded is determined from the set of pages the program had acquired during the last time quantum. This scheme is successful with small time quantum.

The second scheme, proposed by Joseph [JOSE70], is as follows. Whenever a page fault occurs for page x, then fetch page x and x+1. This scheme is particularly successful when the address pattern of a program is sequential.

Prepaging incurs less page faults than demand paging, but the number of page pulls may be the same or even larger because of the possibility of an incorrect prediction [JOSE70]. The effect of prepaging on ST product of MM used is not clear from the literature. Joseph reports an increase in ST product whereas Denning [DENN70] shows that if the probability of incorrect prediction is low, then ST product is reduced by prepaging. Because of a reduction in total page wait time, ST product tends to reduce but at the same time it tends to increase because pages are brought in MM in advance of their use. We will discuss this question in Chapter 3. If more than one page is brought into MM at the same time and a proper layout of pages on the secondary memory is used, then the effective access time per page (T) can be reduced, provided rotating secondary memory is used. The basic question, then seems to be that which performance measure is more important, page fault or page pull. There is no easy answer to this question. However, if the goal is better cpu utilization then page fault is a more important measure and if reduction of the channel traffic is the goal, then page pull is a more important measure. It seems, therefore, that prepaging is justifiable if good predictions are available. We will elaborate on these ideas in subsequent chapters.

2.3 The Formalism of Paging Algorithms

In the abstract, a paging algorithm A is a mechanism for processing a reference string $\omega = r_1 r_2 \ldots r_t \ldots$ and generating, in response, a

sequence of memory states $S_0\ S_1 \ldots S_t \ldots$ where S_0 is a specified initial memory state [AHO71, COFF73]. Each memory state S_t is the set of pages from N which reside in M at time t; they satisfy the conditions $S_t \subseteq N$, $|S_t| \leq c$, $r_t \in S_t$ $(t > 0)$. Moreover, S_t and S_{t-1} are related by, $S_t = S_{t-1} + X_t - Y_t$ where $X_t \subseteq N - S_{t-1}$ is the set of pages fetched at time t and $Y_t \subseteq S_{t-1}$ is the set of replaced pages. To determine X_t and Y_t at time t, the paging algorithm must maintain internal records. A set of control states Q is used for this purpose, q_0 being a designated initial control state. A configuration of the paging algorithm is any pair (S, q) in which $|S| \leq c$, $q \in Q$. Associated with the algorithm A is a transition function g_A, such that $g_A(S, q, x) = (S', q')$, $x \in S'$, where (S, q) is the present configuration and (S', q') is the next configuration and x is the current page reference causing the transition. In particular, the memory state sequence $S_0\ S_1 \ldots S_t \ldots$ induces a configuration sequence $(S_0, q_0)(S_1, q_1) \ldots (S_t, q_t) \ldots$ generated by,

$$(S_t, q_t) = g_A(S_{t-1}, q_{t-1}, r_t),\ t \geq 1.$$

A paging algorithm is a demand fetching algorithm if for a given $c > 0$, $|X_t| \in \{0, 1\}$ and if $r_t \in S_{t-1}$ then $X_t = \emptyset$. A paging algorithm is a demand replacement algorithm if $|Y_t| \in \{0, 1\}$ and if $r_t \in S_{t-1}$ or $|S_{t-1}| < c$ then $Y_t = \emptyset$. A paging algorithm is a demand paging algorithm (demand_fetch_demand_replace) if $g_A(S, q, x) = (S', q')$ satisfies the conditions:

$$S' = \begin{cases} S & \text{if } x \in S \\ S+x & \text{if } x \notin S \text{ and } |S| < c \\ S+x-y & \exists\, y \in S, \text{ if } x \notin S,\ |S| = c. \end{cases}$$

Thus for a demand paging algorithm, $0 \leq |Y_t| \leq |X_t| \leq 1$. A paging algorithm is a demand prefetching algorithm if $g_A(S, q, x) = (S', q')$ satisfies the condition that if $x \in S$ then $X_t = \emptyset$. In particular, the restriction $|X_t| \leq 1$ is removed. And finally, a paging algorithm is a demand prepaging algorithm [COFF73] if $g_A(S, q, x) = (S', q')$ satisfies the conditions:

$$S' = \begin{cases} S & \text{if } x \in S \\ S+X & \text{if } x \notin S,\ x \in X,\ |S+X| \leq c \\ S+X-Y & \text{if } x \notin S,\ x \in X,\ |S+X-Y| = c. \end{cases}$$

In a reference string ω, the forward distance $d_t(x)$ at time t for page x is the distance to the first reference to x after time t:

$$d_t(x) = \begin{cases} k, & \text{if } r_{t+k} \text{ is the first occurrence of } x \text{ in } r_{t+1}, r_{t+2}, \cdots \\ \infty, & \text{if } x \text{ does not appear in } r_{t+1}, r_{t+2}, \cdots \end{cases}$$

Similarly, the backward distance $b_t(x)$ is the distance to the most recent reference to x:

$$b_t(x) = \begin{cases} k, & \text{if } r_{t-k} \text{ is the last occurrence of } x \text{ in } r_1, r_2, \ldots, r_t \\ \infty, & \text{if } x \text{ does not appear in } r_1, r_2, \ldots, r_t \end{cases}$$

Presently, we consider paging algorithms, which work with a fixed memory allotment (i.e., a fixed value of c). Some typical demand paging algorithms will now be described. Let $S_t = S$, and let $S' = S_{t+1} = S_t + x - R(S, q, x)$ such that $x \notin S_t$, $x \in S_{t+1}$ and $R(S, q, x)$ denotes the page selected for replacement so that $R(S, q, x) = 0$ if $|S| < c$.

(1) LRU: This algorithm chooses the page least recently used as the page to be replaced. In other words, $R(S, q, x) = y$ iff $b_t(y) = \max_{z \in S} [b_t(z)]$.

This algorithm is used widely in practice and is found to behave fairly well [DENN70, CORB69, LIPT68].

(2) FIFO: This algorithm chooses for replacement the page that was fetched first in MM. This algorithm behaves well only for highly sequential reference patterns. However, it is very simple to implement.

(3) RAND: It selects the page to be replaced randomly from the pages resident in MM. Clearly, this algorithm behaves well only for independent reference model of program behavior [BELA66, KING71, COFF73]. It is also very simple to implement.

(4) MIN: The page to be replaced has the largest forward distance. In case of a tie, the tie-breaking rule uses lexicographic ordering. This algorithm was first proposed by Belady [BELA66], and it is unrealizable since it presumes the advance knowledge of the program's reference string. It is, however, useful for theoretical purposes since it can be proved optimal in a certain sense. It is also called B_0[COFF73] and OPT[MATT70] sometimes.

More specifically [COFF73], $R(S, q, x) = y$ iff $y = \min_{z \in S^*} [z]$. Here, S^* is defined so that $i \in S^*$ iff $d_t(i) = \max_{u \in S} [d_t(u)]$.

Note here that lexicographical ordering is used in S^*.

2.3.1 Cost of a Paging Algorithm

Aho et al [AHO71] define the cost of processing a reference string $\omega = r_1, r_2, \ldots, r_\gamma$ with a paging algorithm A operating with c page frames of MM: $C(A, c, \omega) = \sum_{t=1}^{\gamma} h(|X_t|)$. Here h is a function such that $h(0) = 0$, $h(k) \geq h(1) = 1$. Aho et al prove the following theorem:

Suppose $h(k) \geq k$ then for any given paging algorithm A, there exists a demand paging algorithm A' such that $C(A', c, \omega) \leq C(A, c, \omega)$ for any reference string ω and for any value of c.

We note that for rotating auxilliary memory devices like discs and drums $h(k) < k$. Only for large core storage device can we hope to have $h(k) = k$. In such a case, the above cost function measures the number of page pulls.

A paging algorithm A is said to be optimal (with respect to the cost function defined) if it minimizes $C(A, c, \omega)$ for all reference strings ω and all values of c. It can be proved that MIN is an optimal paging algorithm by the above definition of optimality [COFF73, POME71, MATT70]. Therefore, the MIN algorithm, though unrealizable, is useful as a benchmark for evaluating other paging algorithms.

Assume that a probability distribution is specified over N^+ such that for $\omega \in N^+$, $p(\omega)$ is the probability of occurrence of ω, then we define the expected cost of a paging algorithm A by:

$$C(A, c) = \sum_{\omega \in N^+} p(\omega)\, C(A, c, \omega) \qquad \text{[COFF73].}$$

Define a paging algorithm to be optimal with respect to distribution p if it minimizes $C(A, c)$, $\forall\, c \geq 1$. Denning et al [DENN68b] proposed a demand paging algorithm A_0, which replaces the page with the longest expected time until next reference. It can be proved that A_0 is an optimal paging algorithm with respect to a distribution corresponding to the independent reference model of program behavior [AHO71].

2.4 Working Set Algorithm

We consider Denning's working set algorithm, which is a variable memory algorithm [DENN68a]. A program's working set at time t is defined to be

$$W(t, \tau) = \{i \in N \mid \text{page } i \text{ appears among } r_{t-\tau+1}, \ldots, r_t\}$$

where τ is a parameter called the window size and $\tau \geq 1$. In other words, $W(t, \tau)$ is the 'contents' of a window of size τ looking backwards at the reference string from reference r_t. Working set replacement algorithm essentially states that, do not replace a page from the working set. This algorithm is a little difficult to implement but behaves very well in practice [DOHE70, RODR72, RODR73, WEIZ69]. Working set principle consists in using a working set replacement algorithm and using the following scheduling policy: a program may run if its working is in MM [DENN70]. Denning has shown that thrashing can be avoided by use of WS principle [DENN68].

Denning et al [DENN72b] define and prove several important properties of the working-set model. Define the working set size $\omega(t, \tau) = |W(t, \tau)|$ and the average working set size $s(\tau) = \lim_{k\to\infty} s_k(\tau)$ where $s_k(\tau) = \frac{1}{k} \sum_{t=1}^{k} \omega(t, \tau)$. Define the binary variable

$$\Delta(t, \tau) = \begin{cases} 1 \text{ if } r_{t+1} \notin W(t, \tau) \\ 0 \text{ otherwise,} \end{cases}$$

then the missing page rate (page fault probability) $m(\tau) = \lim_{k\to\infty} \frac{1}{k} \sum_{t=0}^{k-1} \Delta(t, \tau)$. Suppose that in the reference string ω, two successive references to page i occur at times t and $t+x_i$. We call x_i an interference interval for page i. The interference distribution for page i is defined to be

$$F_i(x) = \lim_{k\to\infty} \left[\frac{\text{no. } x_i \text{ in } r_1, r_2, \ldots, r_k \text{ with } x_i \leq x}{\text{no. } x_i \text{ in } r_1, \ldots, r_k} \right]$$

The interference density function for page i is defined to be $f_i(x) = F_i(x) - F_i(x-1)$. With these definitions following properties hold:

P1: $1 = s(1) \leq s(\tau) \leq s(\tau+1) \leq \min\{n, \tau+1\}$.

P2: $s(\tau+1) - s(\tau) = m(\tau)$.

P3: $0 \leq m(\tau+1) \leq m(\tau) \leq m(0) = 1$.

P4: $m(\tau) = 1 - F(\tau) = \sum_{y<\tau} f(y)$.

P5: $m(\tau+1) - m(\tau) = - f(\tau+1)$.

P6: $s(\tau) = \sum_{z=0}^{\tau-1} m(z) = \sum_{z=0}^{\tau-1} (1-F(z))$

$$= \sum_{z=0}^{\tau-1} \sum_{y<z} f(z).$$

P7: $\lim_{\tau\to\infty} s(\tau) = n$.

P9: $\lim_{\tau\to\infty} m(\tau) = 0$.

We note here that the average working set size of a program is a very good indicator of its locality [DENN70].

2.5 Performance Measurement

If we are given a reference string ω, a page allotment c and a paging algorithm A then we can simulate the processing of ω by A and obtain the page fault count and the page pull count. We can also measure ST product provided the average page wait time is assumed to be fixed. With a demand paging algorithm page faults and page pulls are identical and the ST product is directly dependent on page faults, therefore, most measurements are restricted to only page pulls. As an example, we consider the reference string, ω = ABCDEBCBDAEAC, where N = {A, B, C, D, E}. We use c = 3 and find the page faults for the above ω using LRU.

paging algorithm

	S_1	S_2	S_3	S_4									S_{13}
ω→	A	B	C	D	E	B	C	B	D	A	E	A	C
memory	A	B	C	D	E	B	C	B	D	A	E	A	C
state		A	B	C	D	E	B	C	B	D	A	E	A
S→			A	B	C	D	E	E	C	B	D	D	E
FAULT	*	*	*	*	*	*	*		*	*	*		*

The number of page faults in this example is 11. Observe how the LRU algorithm chooses a page to be replaced. At t=4, r_4=D, $r_4 \notin S_3$ which implies a page fault which in turn implies a replacement. In S_3, A is the page used least recently, therefore, it is chosen for replacement. Now for the same ω and paging algorithm, but for a different value of c, we are asked to find the page fault count, we will have to go through the simulation again. This means a rescan of the reference string for each new value of c. Since practical reference strings are very long, say a million references, the simulation is very inefficient. Mattson et al [MATT70] have developed a technique which scans the reference string only once and produces the page fault count for all values of $1 \leq c \leq n$. We will have occasion to use his ideas in Chapter 3.

2.5.1 Stack Algorithms [COFF73, MATT70]

We introduce the notation S(A, c, ω) to stand for the memory state resulting after the paging algorithm A has processed reference string ω in a memory of size c, assuming that $S_0 = \emptyset$. That is, if $\omega = r_1, r_2, \ldots, r_t$ generates $S_0, S_1, \ldots, S_t$ under A, then $S(A, c, \omega) = S_t$. If A is understood, we simply write S(c, ω).

An algorithm A is called a stack algorithm if its memory states satisfy the following inclusion property: $S(c, \omega) \subseteq S(c+1, \omega)$ $\forall$ $c \geq 1$, $\forall$ $\omega \in N^+$. This means that the memory states form a collection of nested sets. This condition is equivalent to the following condition: for each ω, there exist a permutation of N, $\underline{s}(\omega) = (s_1(\omega), s_2(\omega), \ldots, s_n(\omega))$, such that, for all $1 \leq c \leq n$, $S(c, \omega) = \{s_1(\omega), s_2(\omega), \ldots, s_c(\omega)\}$. The vector $\underline{s}(\omega)$ is called the stack vector or imply the stack. For a given stack algorithm, the inclusion property implies that for each reference string $r_1, \ldots, r_t$, a sequence of stacks $\underline{s}_1, \underline{s}_2, \ldots, \underline{s}_t$ can be constructed so that the memory state sequence for each value of c can be determined by simply taking the topmost c pages of the stack. This property of stack algorithms implies that the page fault behavior of a given reference string can be computed effectively in parallel for all memory sizes $1 \leq c \leq n$ and in one scan of the reference string.

Examples of stack algorithms are MIN, LRU, LFU (least frequently used), whereas FIFO is not a stack algorithm [COFF73, MATT70].

The stack distance $D_x(\omega)$ is defined for page x to be the position of x in the stack $\underline{s}(\omega)$. Thus if $s_k(\omega) = x$, then $D_x(\omega) = k$. If x does not appear in $\underline{s}(\omega)$, then $D_x(\omega) = \infty$. Observe that a page fault occurs for page x as the last reference in the string ωx iff $D_x(\omega) > c$, since the first c elements of $\underline{s}(\omega)$ are the contents of memory of size c. Suppose that $\omega = r_1, r_2, \ldots, r_\gamma$ is processed by A producing the stack sequence $\underline{s}_1, \underline{s}_2, \ldots, \underline{s}_t, \ldots, \underline{s}_\gamma$. There is, associated with ω, a stack distance sequence $D_1, D_2, \ldots, D_\gamma$, where D_t is the position of r_t in $\underline{s}_{t-1}$. If $\pi(A, c, \omega)$ denotes the number of page faults in processing ω with memory size c using algorithm A then,

$$\pi(A, c, \omega) = |\{t | D_t > c, 1 \leq t \leq \gamma|.$$

Define $a_k = |\{t | D_t = k, 1 \leq t \leq \gamma\}|$ then clearly,

$$\pi(A, c, \omega) = a_\infty + \sum_{k=c+1}^{n} a_k.$$

Alternatively, the success function $N(A, c, \omega)$ is $N(A, c, \omega) = \sum_{k=1}^{c} a_k$, the success frequency $F(A, c, \omega) = N(A, c, \omega)/\gamma$, and the page fault frequency $m(A, c, \omega) = \pi(A, c, \omega)/\gamma$.

Gordon [GORD73] has shown a method by which one can compute the average working set size and the page fault probability for the WS algorithm and for the LRU algorithm in one pass of the reference string. Define $I_x(\omega)$ (reference interval) associated with each page in the stack $\underline{s}(\omega)$ as the number of distinct pages referenced since the last reference to this page and $I_x(\omega x) = 1$ (i.e., reference interval is defined to be equal to 1 at the time this page is referenced). If $x \notin \underline{s}(\omega)$ then $I_x(\omega) = \infty$. Let I_t be the reference interval associated with page $r_t = x$ in the stack $\underline{s}_{t-1}$. Then associated with the reference string , there is a reference interval sequence $I_1, I_2, \ldots, I_t, \ldots, I_\gamma$. Clearly $\pi(WS, \tau, \omega) = |\{t | I_t > \tau, 1 \leq t \leq \gamma\}|$. If we define, $b_k = |\{t | I_t = k, 1 \leq t \leq \gamma\}|$, then $\pi(WS, \tau, \omega) = b_\infty + \sum_{k=\tau+1}^{n} b_k$. Also $m(\tau) = \pi(WS, \tau, \omega)/|\omega|$ and $s(\tau) = \sum_{z=0}^{\tau-1} m(z)$ where $m(0) = 1$ by definition.

We have noted that MIN is a stack algorithm but since it requires a scan of the future reference string in order to determine the page to be replaced, a one pass algorithm is difficult to obtain. Mattson et al [MATT70] have given a two pass algorithm to get MIN statistics. More

recently, Belady et al [BELA74] present a one pass algorithm to obtain MIN page fault behavior.

So far we have discussed methods to obtain paging performance when the reference string is specified. Collection of page traces, however, is very time consuming and for very long page traces running simulations to obtain paging performance is also very time consuming. It is beneficial, therefore, to consider analytical methods to predict paging behavior of programs. This requires a model of program behavior to be formulated. King [KING71] has analyzed program behavior using the independent reference model.

2.5.2 King's Model

If the configuration sequence $(S_0, q_0)\ (S_1, q_1) \ldots (S_t, q_t) \ldots$ generated by a paging algorithm form a Markov chain, then analytical methods exist to find m(c) where $m(c) = \sum_{\omega \in N^\infty} p(\omega)\ m(c, \omega)$. Here N^∞ is the set of all infinite reference strings over N.

Suppose we are given the transition probability matrix P of the Markov chain whose states are $\mathbf{V} \times Q$, where $\mathbf{V} = \{S \mid S \subseteq N,\ |S| = c\}$ (i.e., $\mathbf{V}$ is the set of memory states). Now if the Markov chain is irreducible, then there exists a unique stationary probability distribution α such that $\alpha P = P$ and $\sum_{i \in \mathbf{V} \times Q} \alpha_i = 1$. Define the equivalence class of state i of the Markov chain by:

$$[i] = \{i \mid \text{transition from } i \to j \text{ and } j \to i \text{ is a non-page fault transition}\}.$$

It can be shown that,

$$m(c) = 1 - \sum_{j=1}^{V \times Q} \left(\sum_{\{k \mid [k]=[j]\}} \alpha_k \, p_{k,j} \right)$$

where $P = (p_{kj})$. Under the independent reference model of program behavior LRU, FIFO, A_0 algorithms satisfy the criteria for this treatment and closed form expression of page fault probability can be obtained.

3. GENERAL PREPAGING ALGORITHMS

3.0 Introduction

In this chapter we present arguments as to how prepaging can be useful. We present several new prepaging algorithms and study their behavior.

We have noted in Chapter 2 that, Aho et al proved that given an arbitrary paging algorithm A, there exists a demand paging algorithm A' which performs no worse than A. First of all, this theorem holds only if $h(k) \geq k$. For rotating auxilliary memories $h(k) < k$ and only for bulk core storage, do we have $h(k) = k$. Since rotating auxilliary memories are in widespread use, the assumptions of the theorem do not hold in practice. If $h(k) = k$ then the cost function defined by Aho et al measures the number of page pulls. If the channel is the bottleneck in the system then the page traffic should be minimized. Heuristically, minimization of page pulls for individual programs will imply minimization of the system page traffic. Under this assumption the theorem is valid. When cpu overhead is critical [MASU72, ORGA72], however, and we have multiprogramming system then a page fault implies a process switch in which case, how many pages we pull after a page fault is immaterial and therefore the cost of a paging algorithm is the number of page faults it incurs. Under this assumption, $h(k) = 1 \ \forall \ k \geq 1$. Reduction in cpu overhead is affected in two ways: first since the number of page faults is decreased and as we have seen, few milliseconds of cpu time is spent

in servicing a page fault. Secondly, after a page fault, if a ready process is not available, the cpu may have to be idle. By reducing the number of page faults we reduce the possibility of such idleness. The effect of prepaging on ST factor will be discussed in section 3.1.

Granted that prepaging has its benefits, there are two problems yet to be resolved: who should specify the paging needs in advance and when to carry out the page fetches. Haore [HAOR72] notes that the memory management routines do not have sufficient knowledge of the future reference strings, therefore, these forecasts must be made by the programs, either directly by the programmer or by the compiler. We agree with this statement. To show the viability of prepaging, we will consider a particular application and show that prepaging can be done and that it does result in improved performance. If we resort to demand prepaging, then the problem of when to fetch pages is solved. Whenever a prepage request is made, we just flag the page and on the occurrence of a page fault, we bring in all the flagged pages.

Note that, under demand paging, the distinction between a page fault and a page pull vanishes since either implies the other. Under an arbitrary paging algorithm neither need imply the other. Under demand prepaging, a page fault implies a page pull but not vice versa. This means that for demand prepaging, # page faults $\leq$ # page pulls. The ideal for the number of page faults is zero. It is possible, at least in theory, to achieve this ideal with arbitrary prepaging but not with demand prepaging or with demand paging.

3.1 The Optimal Demand Prepaging Algorithm

We define a demand prepaging algorithm DPMIN using the notation of section 2.3.

DPMIN: The transition function $g(S, q, x) = (S', q')$

where,

$S' = S$ if $x \in S$

$S' = \{y_1, y_2, \ldots, y_\ell\}$ if $x \notin S$

where $\ell = \min(c, |{}_t\omega|)$ and

${}_t\omega = r_t, \ldots, r_\gamma$, and where

$\forall i, y_i \in N \;\&\; \forall x \in N - S' \quad d_t(x) > d_t(y_i)$.

In other words, at the time of a page fault, DPMIN scans the future reference string and fetches the first c pages that will be referenced in future.

Note that, DPMIN is unrealizable in the same sense as MIN is unrealizable. We will see that DPMIN serves as a benchmark of performance.

We define a paging algorithm A to be optimal with respect to page faults if for any arbitrary paging algorithm A', and $\forall c \geq 1$, $\forall \omega \in N^*$,

$$\pi(A, c, \omega) \leq \pi(A', c, \omega),$$

where π denotes the number of page faults.

Theorem 3.1:

DPMIN is optimal among all demand prepaging algorithms (in number of page faults).

Proof: Let $\pi_k(S, t)$ denote the minimum achievable cost under demand prepaging of processing references $r_{t+1}, \ldots, r_{t+k}$ given that $S_t = S$. If

we define $\pi_0(S, t) = 0$ and let $r_{t+1} = x$ then we can write:

$$\pi_k(S, t) = \begin{cases} \pi_{k-1}(S, t+1) & \text{if } x \in S \\ 1 + \min\limits_{\substack{Y \subseteq S \\ X \subseteq N-S \\ x \in X}} \pi_{k-1}(S+X-Y, t+1) & \text{if } x \notin S. \end{cases}$$

This relation may be recognized as the principle of optimality in a dynamic programming problem [AHO71, COFF73]. The proof of the theorem now reduces to showing that paging according to DPMIN is characterized by the above principle of optimality.

Let $<_t$ be an ordering defined over N for $\forall\, t \geq 1$ such that $x <_t y$ iff $d_t(x) \leq d_t(y)$ (if $d_t(x) = d_t(y)$, lexicographic ordering is assumed). Denote $M_t = \{y_1, \ldots, y_{c-1}\}$ such that $\forall\, x \in N-M_t$, $x < y_i$ for $\forall\, 1 \leq i \leq c-1$. Then the transition function for DPMIN may be written as:

$$g_{DPMIN}(S, q, x) = \begin{cases} (S, q') & \text{if } x \in S \\ (M_t+x, q') & \text{if } x \notin S. \end{cases}$$

Then it is sufficient to show that

$$\pi_k(M_t+x, t-1) = \min_{\substack{S \subseteq N-x \\ |S|=c-1}} (\pi_k(S+x, t-1)).$$

Clearly, $\pi_k(M_t+x, t-1) = \pi_{k-1}(M_t+x, t)$ and

$$\pi_k(S+x, t-1) = \pi_{k-1}(S+x, t),$$

therefore, it is sufficient to show that

$$\pi_k(M_t+x,\ t) = \min_{\substack{S\subseteq N-x \\ |S| = c-1}} (\pi_k(S+x,\ t)).$$

In fact, we will prove that $\Delta\pi_k = \pi_k(S+x,\ t) - \pi_k(M_t+x,\ t) \leq 1$. We will prove this by induction on k. It is clearly true for k = 0 (by definition of $\pi_k(S,\ t)$). Assume true for $\forall\ i < k$. If $S = M_t$ then we are done, therefore assume $S \neq M_t$ and let i be the smallest index such that $y_i \notin S$, also let $d_t(y_i) = \ell$. First note that $i < c-1$. Then,

$$\pi_k(S+x,\ t) = 1 + \min.(\pi_{k-\ell}(S+X-Y,\ t+\ell+1))$$

and

$$\pi_k(M_t+x,\ t) = \pi_{k-\ell}(M_t+x,\ t+\ell+1).$$

By inductive assumption,

$$\min.(\pi_{k-\ell}(S+X-Y,\ t+\ell+1))$$

$$= \pi_{k-\ell}(M_{t+\ell}+r_{t+\ell+1},\ t+\ell+1)$$

Also, by induction,

$$\Delta\pi_{k-\ell} = \pi_{k-\ell}(S'+r_{t+\ell+1},\ t+\ell+1)$$

$$- \pi_{k-\ell}(M_{t+\ell}+r_{t+\ell+1},\ t+\ell+1) \leq 1.$$

Then,

$$\Delta\pi_k = \pi_k(S+x,\ t) - \pi_k(M_t+x,\ t)$$

$$= 1 + \pi_{k-\ell}(M_{t+\ell}+r_{t+\ell+1},\ t+\ell+1)$$

$$- \pi_{k-\ell}(M_t+x,\ t+\ell+1)$$

note that $r_{t+\ell+1} \in M_t+x$.

$\therefore$ taking $S' = M_t+x-r_{t+\ell+1}$,

$$\Delta\pi_k = 1 + \pi_{k-\ell}(M_{t+\ell}+r_{t+\ell+1},\ t+\ell+1) - \pi_{k-\ell}(S'+r_{t+\ell+1},\ t+\ell+1)$$

$= 0$ or 1 by the inductive hypothesis on $\Delta\pi_{k-\ell}$.

Therefore for $\forall\ k \geq 0$,

$$\Delta\pi_k \leq 1$$

This completes the proof of the optimality of DPMIN.

Lemma 3.1:

Given two fixed memory paging algorithms A and A'

$$\pi(A,\ c,\ \omega) < \pi(A',\ c,\ \omega) \Rightarrow ST(A,\ c,\ \omega) < ST(A',\ c,\ \omega)$$

on the average, assuming the behavior of other concurrent programs remain the same.

Proof: Since memory alloted to the program is fixed, $ST = c * t_m$ where t_m is the total time that the program occupies the MM. $t_m \simeq t_{cpu} + t_{page\text{-}wait} + t_{ready\ queue}$. Since cpu time of a program is unaltered by a change in the paging algorithm, only the other two factors need to be considered. Now assume T' is the average time that the program has to spend in page wait and then in the ready queue. Clearly, T' depends on the characteristics of other concurrently executing programs and on the average page fetch time T. Therefore, T' can be assumed to be a constant.

$$t_m = t_{cpu} + \#\text{page faults} * T'$$

The required result immediately follows.

Theorem 3.2:

DPMIN minimizes ST product among all demand prepaging algorithms under the assumptions of lemma 3.1.

Proof: The proof of this theorem follows directly from theorem 3.1 and lemma 3.1.

We would have liked to prove optimality of DPMIN with respect to page pulls, but unfortunately, it does not hold. We will be able to see this in Chapter 5. As a result of lemma 3.1, we need to only discuss page fault and page pull measures for all fixed memory type paging algorithms.

Note that the class of all demand paging algorithms is included in the class of all demand prepaging algorithms. Therefore, applying Theorem 3.1 to the optimal demand paging algorithm MIN, we get

$$\pi(\text{DPMIN}, c, \omega) \leq \pi(\text{MIN}, c, \omega).$$

We will show that there are cases when strict inequality holds. On the other hand, it is trivial to find examples for which equality holds. Note that MIN is optimal in page pulls among all paging algorithms, therefore,

$$\begin{aligned} C(\text{DPMIN}, c, \omega) &\geq C(\text{MIN}, c, \omega) \\ &= \pi(\text{MIN}, c, \omega). \end{aligned}$$

Once again, there are cases when equality holds in the above relation and there are cases when strict inequality holds. Thus DPMIN is superior in two performance measures (page faults and ST product) and MIN is superior in page pulls. Quantitative results of the comparison will be given in Chapter 5 for specific applications.

3.1.1 Performance Measurement for DPMIN

The problem is to find the number of page faults and the number of page pulls for a given reference string ω and a given page allotment c, using the DPMIN paging algorithm. We present a numeral matrix algorithm for this purpose, based on the numeral matrix algorithm for MIN [BELA74]. The numeral matrix has as many rows as the number of pages in the VA space and as many columns as the nu er of page faults. The algorithm is as follows:

Initially the matrix is blank.

1. Suppose the next reference in ω is to a page x and that the rightmost nonempty column is (i-1).
2. Let j be the rightmost column with c markings (note: j = i-1 or i-2).
3. If j = i-1
   then if the entry (x, i-1) is blank
   then mark (x, i) and return
   else return
   else
4. If j = i-2 then
   if (x, j) is blank then mark (x, i-1)
   and return else mark (x, i-1) and return.

In step 3 marking (x, i) implies a page fault and a page pull. In step 4 if (x, j) is blank then a page pull occurs (no page fault).

After the complete reference string is processed, then the number of page faults is obtained by counting the number of nonempty columns of the numeral matrix. Number of page pulls in excess of page faults is obtained by counting the number of times the clause 'if (x, j) is blank' is satisfied in step 3.

Clearly, this algorithm is c-dependent. We illustrate its use by the following example.

N = {A, B, C, D, E}, c = 3,

ω = ABCDEDBCBDAEAC. The number 1 is used for marking. If a page pull occurred without a page fault then we indicate this with an asterisk. The numeral matrix for our example is:

A	1			1
B	1*	1	1	
C	1*		1	1
D		1	1	
E		1*		1*

Clearly, the number of page faults is equal to four. The number of page pulls is given by:

page pulls = # of 1* + # page faults

= 4 + 4 = 8.

Belady et al [BELA74] show that for the same reference string, the MIN algorithm produces

page faults = 8 = # of page pulls.

The numeral matrix algorithm is clearly c-dependent, i.e., for each new value of c, a rescan of the reference string is needed. In particular, DPMIN is not a stack algorithm. We may verify this fact by the following example: ω = ABCD, for c = 2, the memory state sequence is: {}, {A, B}, {A, B}, {C, D}, {C, D}. For c = 3, the memory state sequence is: {}, {A, B, C}, {A, B, C}, {A, B, C}, {D}. From these, $S_3(2) \not\subseteq S_3(3)$ which gives the required result.

Another problem in the implementation of the numeral matrix algorithm is the storage of the matrix. A close inspection, however, reveals that only two columns of the numeral matrix need be stored at any one time. In step 2, we have noted that j = i-1 or i-2. This is because, we create a new column only when the previous column is full. Thus only the present column and the last column need be kept in storage. A PL/I implementation of DPMIN is given in Appendix A.

3.2 Realizable Prepaging Algorithms

We have noted that DPMIN is an unrealizable paging algorithm. We would like to consider some paging algorithms which can be realized.

3.2.1 Freeing Dead Pages

Assume that at time t, a certain page x is 'dead', i.e., it does not occur in the reference string after time t. This is equivalent to saying: $d_t(x) = \infty$. Assume that either the programmer or the compiler has discovered this fact. Furthermore assume that this information is provided to the operating system so that either it can push the page x onto the secondary memory or it can flag the page dead so that at the time of the next replacement decision, page x will have the highest priority to be replaced. Assume that, originally, we started out with a demand paging algorithm A, and then modify it by the above mechanism and call it FREEA. The process of declaring a page dead will be called Freeing. There are two possible ways to interpret FREEA: One way is to assume that on the issuance of a FREE(x) instruction, page x is replaced. In this case FREEA is a demand-fetch-anticipatory-replace algorithm. The second way is to assume that on the issuance of a FREE(x) instruction, page x is

flagged (i.e., a 'dead' bit associated with the page table entry of page x is set), and when a replacement decision is to be made, first a search is made for a page with the dead bit on. If none is found then the replacement procedure as in algorithm A is followed. With this interpretation, FREEA is a demand paging algorithm. Quite clearly, these two interpretations do not affect the number of page pulls or the number of page faults and therefore, we do not have to distinguish between them. It is clear that $\pi(\text{FREEA}) \leq \pi(\text{A})$. To verify this, assume that A has made a wrong replacement decision at time t, and further assume that working with FREEA, at time t, a dead page was present in memory and therefore FREEA will not make a wrong replacement decision. On the other hand, if FREEA has made a wrong replacement decision then quite clearly, A must have also.

It is also clear that $\pi(\text{MIN}) \leq \pi(\text{FREEA})$, since FREEA is a demand paging algorithm and the optimality of MIN is applicable. It should be noted that $\pi(\text{MIN}) = \pi(\text{FREEMIN})$, since MIN never makes a wrong replacement decision. If we started out with a demand prepaging algorithm A then we can show that

$$\pi(\text{DPMIN}) \leq \pi(\text{FREEA}) \leq \pi(\text{A})$$

and that $$\pi(\text{DPMIN}) = \pi(\text{FREEDPMIN}).$$

If we start out with a general paging algorithm A, then $0 \leq \pi(\text{FREEA}) \leq \pi(\text{A})$. Freeing, however, tends to reduce the ST product: First by reducing the number of page faults and second by reducing the average page-wait time. The second reason holds only when anticipatory replacement is used.

We have pointed out in Chapter 2 that, it is desirable that a given paging algorithm is a stack algorithm. Theorem 3.3 answers this question for FREEA.

Theorem 3.3: (Assume A is a demand paging algorithm)

If A is a stack algorithm then FREEA is a stack algorithm.

Proof: The proof makes use of the proposition 6.1 from the book by Coffman et al [COFF73] which states that:

A demand paging algorithm B is a stack algorithm

iff $R(S+y, q, x) = R(S, q, x)$ or y whenever $x \notin S + y$.

Let Γ_t denote the set of distinct pages in the string $r_1, r_2, \ldots, r_t$. FREEA partitions the pages of Γ_t into two classes. One class is the set of dead pages. We assume that the set of dead pages is lexicographically ordered, and the set of nondead pages is ordered by the same ordering as in algorithm A. Let the set of dead pages in S be denoted by $\Delta(S)$. We want to prove that the above proposition is satisfied by FREEA. Therefore, consider the following two cases:

(i) $[y \in \Delta(S+y)]$:

If $y = \min.(\Delta(S+y))$ then $R(S+y, q, x) = y$

and we are done, otherwise

if $y \neq \min.(\Delta(S+y))$ then let

$y' = \min.(\Delta(S+y))$. This means

$y' = \min.(\Delta(S))$ which implies

$R(S+y, q, x) = R(S, q, x) = y'$

and we are done.

(ii) $[y \notin \Delta(S+y)]$:

If $\Delta(S) = \emptyset$ then $\Delta(S+y) = \emptyset$ (since $y \notin \Delta(S+y)$)

therefore, FREEA behaves exactly like A

which means $R(S+y, q, x) = R(S, q, x)$

(since A is a stack algorithm).

If $\Delta(S) \neq \emptyset$ then let $y' = \min.(\Delta(S))$.

Since $y \notin \Delta(S+y) \Rightarrow \Delta(S) = \Delta(S+y)$

$\Rightarrow y' = \min.(\Delta(S+y)) \Rightarrow$

$R(S+y, q, x) = R(S, q, x) = y'$.

Thus FREEA is a stack algorithm.

In Appendix A, we give a PL/I program for FREELRU.

3.2.2 Prepaging

We now consider some practical prepaging algorithms. Our objective is to consider prepaging algorithms which reduce page faults without increasing page pulls significantly and thereby obtain an improvement over demand paging algorithms. When working with fixed memory algorithms, at the beginning of a time slice, a few page frames remain unoccupied with a demand paging situation. If we could prepage some useful pages to fill up this unused space, we would probably reduce page faults. The second important rule we incorporate into the algorithm is that we never replace a useful (non-dead) page for bringing in a prepaged page. This reduces the possibility of increasing the page faulting in a prepaging algorithm After the MM is full, this scheme would not allow us to do any prepaging unless, we have some dead pages which can be freed. We will see that this situation is the most valuable.

The first prepaging algorithm that we propose is PRE1A. Assume that reference string is modified to have PRE(x) inserted at various places.

PRE1A: (Assume A is a demand paging algorithm)

If $|S_t| < c$ then if $r_{t+1} = PRE(x)$

and $x \notin S_t$ then $S_{t+1} = S_t + x$.

Otherwise PRE1A works just like A.

This algorithm has two drawbacks. First, it implies a page-wait for each prepage operation which is what we were trying to avoid in the first place and second, it is not able to prepage frequently since the situation $|S_t| < c$ arises only initially. We can solve the first problem in two ways. One way is to allow the execution of the program and the input operation of the prepaged page simultaneously, which will be called the "overlap solution." A second way is to concatenate many page fetches together and resort to demand prepaging. The solution to '$|S_t| < c$' problem is to combine freeing and prepaging together.

If we use the overlap solution then after the occurrence of PRE(x) in the reference string, a page frame is reserved for x but x cannot be assumed to reside in S_{t+1}. Page x is then said to be in a 'not-set-up' state. After elapsing of real time T (the average page-wait time), page x said to be in 'set-up' state and can be considered resident in MM. Now, if before a page x is set up, a page fault for another page y (or the same page x) occurs, then during this page-wait period, page x will become set-up. Thus, in general, on the occurrence of a page fault, all not-set-up pages in MM will become set-up. We divide the memory state $S_t(c)$ into two disjoint sets $U_t(c)$ and $N_t(c)$. $N_t(c)$ consists of the set of all not-set-up pages and $U_t(c) = S_t(c) - N_t(c)$. We will assume $c \geq 2$ for all prepaging algorithms. We now define PRE2A. We associate a set-up counter 'COUNT' with each not-set-up page.

PRE2A:

[1] (Page fault for a not-set-up page)

If $r_{t+1} = x \in N_t(c)$ then (declare all not-set-up pages set-up)

$U_{t+1}(c) = U_t(c) \cup N_t(c) = S_t(c) = S_{t+1}(c)$.

$N_{t+1}(c) = \emptyset$.

We can easily add freeing to PRE2A and obtain FREEPRE2A. It can easily be proved that both, PRE2A and FREEPRE2A are not stack algorithms.

It appeared at first that if we restrict our attention to a certain class of reference strings then PRE2A could be proved to be a stack algorithm. We will say that a reference string ω satisfies the property P if $r_t = PRE(x) \Rightarrow (\nexists\, t' < t$ such that $r_{t'} = x$ or $r_{t'} = PRE(x))$.

Initially it was believed that PRE2A is a stack algorithm for $\{\omega \in N^+ \mid P(\omega)\}$, however, this was found to be false. The reason for this can be explained as follows: Suppose we are in step (3) of PRE2A. For a certain value of c, say c_1, assume that $r_{t+1} = x \notin S_t(c_1)$ and let c_1 be the largest such value of c, $\Rightarrow x \in S_t(c_1+1)$ (assume that x is set-up in a memory of size c_1+1). A page fault occurs for $c = c_1$, whereas no page fault occurs for $c = c_1+1 \Rightarrow U_{t+1}(c_1) = S_t(c_1) + x - y$, $y \in U_t(c_1)$. But, $U_{t+1}(c_1+1) = U_t(c_1+1) \cup \{y \in N_t(c_1+1),\ COUNT(y) \geq T-1\}$. Now, the inclusion property for U_t cannot be proved in general. A similar situation occurs in step (1). We, therefore, modify PRE2A to obtain PRE3A, which is indeed a stack algorithm.

PRE3A:

[1] If $r_{t+1} = x \in N$ then

[a] if $x \in N_t(c)$ then (considered to be a page fault)

$\forall\, y \in N_t(c)$, $COUNT(y) \leftarrow COUNT(y) + 1$;

$U_{t+1}(c) = U_t(c) + x \cup \{y \mid y \in N_t(c),\ COUNT(y) \geq T\}$;

$S_{t+1}(c) = S_t(c)$;

$N_{t+1}(c) = S_{t+1}(c) - U_{t+1}(c)$;

and RETURN;

[2] (Page reference to a set-up page)

If $r_{t+1} = x \in U_t(c)$ then

(increment COUNT for each non-set-up page)

$\forall\, y \in N_t(c)$ $\quad$ COUNT(y) $\leftarrow$ COUNT(y) + 1.

(declare all non-set-up pages with COUNT $\geq$ T, set-up)

$U_{t+1}(c) = U_t(c) \cup \{y \in N_t(c) \mid \text{COUNT}(y) \geq T\}$.

$S_{t+1}(c) = S_t(c)$

$N_{t+1}(c) = S_{t+1}(c) - U_{t+1}(c)$.

[3] (reference to page not in $S_t(c) \Rightarrow$ page fault)

If $r_{t+1} = x \notin S_t(c)$ then

(bring in the required page, replace a page if necessary and declare all not-set-up pages set-up).

$U_{t+1}(c) = S_t(c) + x - R_A(U_t, q, x)$

$N_{t+1}(c) = \varphi$, $S_{t+1}(c) = U_{t+1}(c)$.

Note here that $R_A(U_t, q, x)$ denotes the page that would have been replaced by paging algorithm A with the given values of the parameters.

[4] (a prepage instruction)

If r_{t+1} = PRE(x) and $x \notin S_t(c)$ then

if $|S_t| < c$ then

$N_{t+1}(c) = N_t(c) + x$,

COUNT(x) $\leftarrow$ 0;

$U_{t+1}(c) = U_t(c)$.

End PRE2A.

[b] If $x \in U_t(c)$ then (reference to a set-up page)

$\forall\, y \in N_t(c)$ COUNT(y) $\leftarrow$ COUNT(y) + 1;

$U_{t+1}(c) = U_t(c) \cup \{y | y \in N_t(c),\ \text{COUNT}(y) \geq T\}$;

$S_{t+1}(c) = S_t(c)$;

$N_{t+1}(c) = S_{t+1}(c) - U_{t+1}(c)$

and RETURN;

[c] If $x \notin \Gamma_t$ (equivalent to saying $x \notin S_t(c')$ for any c' which is equivalent to saying $\nexists\, t' \leq t$ such that $r_{t'} = x$ or $r_{t'} = \text{PRE}(x)$ which is equivalent to saying $x \neq \underline{s}_t(k)$ for any k) then

if $|S_t(c)| < c$ then

$U_{t+1}(c) = S_t(c) + x$; $N_{t+1}(c) = \varphi$;

$S_{t+1}(c) = U_{t+1}(c)$;

else $U_{t+1}(c) = S_t(c) + x - R_A(U_t, q, x)$;

$N_{t+1}(c) = \emptyset$; $S_{t+1}(c) = U_{t+1}(c)$;

and RETURN;

[d] If $x \notin S_t(c)$ [but $x \in \Gamma_t$] then

$\forall\, y \in N_t(c)$ COUNT(y) $\leftarrow$ COUNT(y) + 1;

if $|S_t(c)| < c$ then

$U_{t+1}(c) = U_t(c) + x \cup \{y | y \in N_t(c),\ \text{COUNT}(y) \geq T\}$;

$S_{t+1}(c) = S_t(c) + x$; $N_{t+1}(c) = S_{t+1}(c) - U_{t+1}(c)$;

and RETURN;

else $(|S_t(c)| = c)$

$U_{t+1}(c) = U_t(c) + x - R_A(U_t, q, x)$

$\cup \{y | y \in N_t(c),\ \text{COUNT}(y) \geq T\}$;

$S_{t+1}(c) = S_t(c) + x - R_A(U_t, q, x)$;

$N_{t+1}(c) = S_{t+1}(c) - U_{t+1}(c)$;

and RETURN;

else RETURN;

[2] If r_{t+1} = PRE(x), x ∈ N and x ∉ $S_t(c)$

then

if $|S_t(c) < c$ then $N_{t+1}(c) = N_t(c) + x$;

$U_{t+1}(c) = U_t(c)$; COUNT(x) ← 0;

$S_{t+1}(c) = S_t(c) + x$;

and RETURN;

END PRE3A;

We will say that an absolute page fault has occurred at time t+1 if r_{t+1} = x and x ∉ Γ_t (or any of the equivalent conditions). What this amounts to is that r_{t+1} = x is the first occurrence of x in ω. PRE3A is different from PRE2A in that, only on the occurrence of an absolute page fault, all not-set-up pages are declared set-up, whereas in the latter on every page fault this is done.

Theorem 3.4:

PRE3A is a stack algorithm for $\{\omega | \omega \in N^+, P(\omega)\}$.

Proof: See Appendix B for the proof.

Theorem 3.4 allows us to construct an efficient one pass algorithm to obtain the paging performance of a given reference string ω. Because of the nature of PRE3A, we would expect PRE2A to give better results than PRE3A. Intuitively, it seems that π(PRE2A, c, ω) ≤ π(PRE3A, c, ω), though we have not been able to prove this relation. However, results of applying PRE3A to a certain reference string does tell us, at least approximately, the performance of PRE2A. We note that for each of the algorithms PREiA we can create another algorithm, say FREEPREiA, which

incorporates freeing and thereby improve the paging performance. Clearly, FREEPRE3A is a stack algorithm which follows from theorem 3.3.

In both, PRE2A and PRE3A, whenever a page made a transition from not-set-up state to set-up state, we immediately consider it the same as any other page in U_t. This means that a prepaged page can be replaced even before it is used at least once. This kind of replacement may lead to increased page traffic. When a PRE(x) has been inserted in the reference string, either by the programmer or by the compiler, we have reason to believe that page x will be used in the near future, so it might pay off to keep the page locked in MM until its first use. We now, divide the memory state $S_t(c)$ into three disjoint sets: $U_t(c)$ is the set of used pages (used at least once), $P_t(c)$ is the set of prepaged, set-up but not yet used pages and $N_t(c)$ is the set of not-set-up pages. We define PRE4A corresponding to PRE2A as follows:

PRE4A:

[1] If $r_{t+1} = x \in N$ then

[a] if $x \in N_t(c)$ then (page fault)

$U_{t+1}(c) = U_t(c) + x$

$P_{t+1}(c) = P_t(c) + N_t(c) - x$

$N_{t+1}(c) = \emptyset$ and RETURN;

[b] if $x \in P_t(c)$ then (success)

$\forall\, y \in N_t(c)$ COUNT(y) = COUNT(y) + 1;

$P_{t+1}(c) = P_t(c) - x \cup\{y \mid y \in N_t(c),\ \text{COUNT}(y) \geq T\}$;

$U_{t+1}(c) = U_t(c) + x$;

$N_{t+1}(c) = \{y \in N_t(c) \mid \text{COUNT}(y) < T\}$;

and RETURN;

[c] if $x \in U_t(c)$ then (success)

$\forall\, y \in N_t(c)$, COUNT(y) $\leftarrow$ COUNT(y) + 1;

$U_{t+1}(c) = U_t(c)$;

$P_{t+1}(c) = P_t(c) \cup \{y \mid y \in N_t(c),\ \text{COUNT}(y) \geq T\}$;

$N_{t+1}(c) = \{y \in N_t(c) \mid \text{COUNT}(y) < T\}$;

and RETURN;

[d] if $x \notin S_t(c)$ then (page fault)

$U_{t+1}(c) = U_t(c) + x - R_A(U_t, q, x)$;

$P_{t+1}(c) = P_t(c) \cup N_t(c)$;

$N_{t+1}(c) = \emptyset$;

and RETURN;

[2] If $r_{t+1} = \text{PRE}(x)$ then if $x \notin S_t$ then

if $|S_t(c)| < c$ then

$N_{t+1}(c) = N_t(c) + x$; COUNT(x) $\leftarrow$ 0;

$P_{t+1}(c) = P_t(c)$; $U_{t+1}(c) = U_t(c)$;

END PRE4A;

Quite clearly, PRE4A is not a stack algorithm. We can define another algorithm PRE5A which does a not-set-up to set-up conversion only on the occurrence of an absolute page fault. PRE5A will be proved to be a stack algorithm.

PRE5A:

[1] If $r_{t+1} = x \in N$ then

[a] if $x \in N_t(c)$ then (a page fault)

$\forall\, y \in N_t(c)$, COUNT(y) $\leftarrow$ COUNT(y) + 1;

$P_{t+1} = P_t \cup \{y \in N_t\text{-}x \mid \text{COUNT}(y) \geq T\}$;

$U_{t+1} = U_t + x$;

$N_{t+1} = \{y \in N_t\text{-}x \mid \text{COUNT}(y) < T\}$;

and RETURN;

[b] if $x \in P_t(c)$ then (success)

$\forall\, y \in N_t$, COUNT(y) $\leftarrow$ COUNT(y) + 1;

$P_{t+1} = P_t - x \cup \{y \in N_t \mid \text{COUNT}(y) \geq T\}$;

$U_{t+1} = U_t + x$;

$N_{t+1} = \{y \in N_t \mid \text{COUNT}(y) < T\}$;

and RETURN;

[c] if $x \in U_t$ then (success)

$\forall\, y \in N_t$, COUNT(y) $\leftarrow$ COUNT(y) + 1;

$U_{t+1} = U_t$;

$P_{t+1} = P_t \cup \{y \in N_t \mid \text{COUNT}(y) \geq T\}$;

$N_{t+1} = \{y \in N_t \mid \text{COUNT}(y) < T\}$;

and RETURN;

[d] if $x \notin \Gamma_t$ then (an absolute page fault)

$U_{t+1} = U_t + x - R_A(U_t, q, x)$;

$P_{t+1} = P_t \cup N_t$;

$N_{t+1} = \Phi$

and RETURN;

[e] if $x \notin S_t$ then (a page fault)

$\forall\, y \in N_t$, COUNT(y) $\leftarrow$ COUNT(y) + 1;

$U_{t+1} = U_t + x - R_A(U_t, q, x)$;

$P_{t+1} = P_t \cup \{y \in N_t \mid \text{COUNT}(y) \geq T\}$;

$N_{t+1} = \{y \in N_t \mid \text{COUNT}(y) < T\}$;

and RETURN;

[2] if r_{t+1} = PRE(x) and $x \notin S_t$ and

$|S_t(c)| < c$ then (prepage the required page)

$U_{t+1} = U_t$

$N_{t+1} = N_t + x$; COUNT(x) $\leftarrow$ 0;

$P_{t+1} = P_t$; and RETURN;

END PRE5A;

Theorem 3.5:

PRE5A is a stack algorithm for $\{\omega \in N^+ | P(\omega)\}$.

The proof is very similar to the proof of theorem 3.4 so we omit it here.

When faced with the choice between the use of PRE4A and PRE5A, we recommend the use of PRE4A. However, for efficiency in performance measurement, one could use PRE5A. In the following theorem we prove that the performance of PRE4A is no worse than the performance of PRE5A which explains the comments just made about these two algorithms.

Theorem 3.6:

$\pi(\text{PRE4A}, c, \omega) \leq \pi(\text{PRE5A}, c, \omega)$.

Proof: We will first prove that $\forall\ t \geq 1$,

$$S_t(\text{PRE4A}, c, \omega) = S_t(\text{PRE5A}, c, \omega),$$

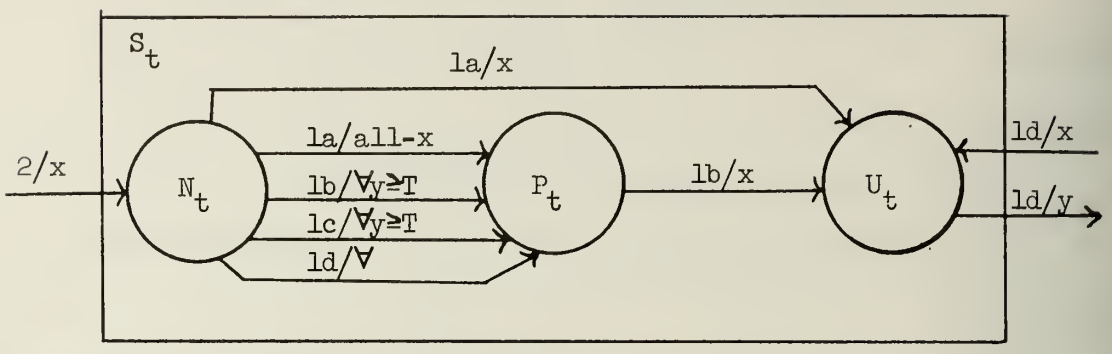
$$U_t(\text{PRE4A}, c, \omega) = U_t(\text{PRE5A}, c, \omega),$$

$$P_t(\text{PRE4A}, c, \omega) \supseteq P_t(\text{PRE5A}, c, \omega).$$

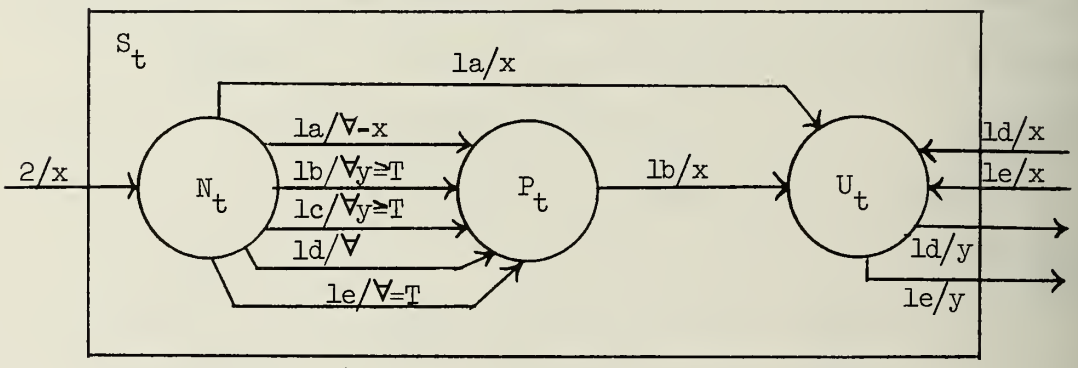
From these, the required result follows since a page fault occurs when $r_{t+1} = x \in S_t(c)$ but $x \notin P_t(c)$, $x \notin U_t(c)$.

The proof is obtained by studying the inputs and outputs to the sets P_t, U_t, N_t. We draw a set input-output diagram in which three circles are the sets P_t, N_t, U_t. The arcs are the input and outputs to these sets. Labels to the arcs are of the form d/X where d is the step of the algorithm in which this arc is relevant and X is the set of pages to which this arc applies.

For PRE4A, the diagram is



For PRE5A, the diagram is:



Note that the following relation between the steps of the two algorithms holds:

PRE4A. (1d) = PRE5A. (1d) ∪ PRE5A. (1e).

and PRE4A. (2) = PRE5A. (2).

$\Rightarrow$ PRE4A. S_t = PRE5A. S_t $\quad \forall\, t \geq 1, \forall\, c \geq 2.$

Input to U_t is strictly a function of ω and output from U_t occurs only in steps 1(d) and 1(e) therefore by the above step relations,

PRE4A. U_t = PRE5A. U_t $\forall\, t \geq 1, \forall\, c \geq 2.$

Then clearly, by the arrows going from N_t to P_t we conclude that

$$\text{PRE}^4\text{A}.\ P_t \supseteq \text{PRE5A}.\ P_t \qquad \forall\, t \geq 1,\ \forall\, c \geq 2.$$

This gives the required result. □

Note that, freeing can be added to any PREdA d = 1, 2, 3, 4, 5 and we obtain the corresponding FREEPREdA with a better performance, i.e., $\pi(\text{PREdA}) \geq \pi(\text{FREEPREdA})$. We can also prove that FREEPRE3A and FREEPRE5A are stack algorithms for $\{\omega \in N^+ \mid P(\omega)\}$. We can also show that $\pi(\text{FREEPRE}^4\text{A},\ c,\ \omega) \leq \pi(\text{FREEPRE5A},\ c,\ \omega)$ which means that, though FREEPRE^{4}A is recommended for use in practice, performance can be bounded by that of FREEPRE5A in lesser time.

Note that each of these prepaging algorithms can be reduced to a demand prepaging algorithm by removing the mechanism of time-dependent set-up of a not-set-up page and allow set-up only on the occurrence of a page fault. To implement this in practice, we do the following: on the occurrence of a PRE(x) in ω, a prepage bit is turned on in the page table entry associated with x. At the time of the next page fault, say, for a page y, we issue page pull commands for y and all the pages with prepage bit turned on (subject to the memory size constraint). We will label (FREE)PREdA, after the above transformation as (FREE)DPREdA. We note that,

$$\pi(\text{DPMIN},\ c,\ \omega) \leq \pi(\text{FREEDPREdA},\ c,\ \omega).$$

We now have to show that these new paging algorithms do achieve something for us in practice. We will use FREELRU and FREEDPRE^{4}LRU and study the behavior of these two algorithms as compared to LRU, MIN and DPMIN. We will carry out our experiments on several matrix problems in Chapter 5 and report the results in that chapter.

3.2.2.1 Joseph's OBL Algorithm [JOSE70]

So far, we have only discussed deterministic prepaging, i.e., the prepaging specified by the programmer or compiler. In the deterministic case, we assumed that the prediction is always correct. Prepaging schemes developed by Joseph [JOSE70] can be called probabilistic prepaging schemes. These schemes are implemented in the operating systems and does not need a prescan of the program. They are based on the observation that a program tends to refer to its pages sequentially. Joseph has shown experimentally that such prepaging schemes do reduce page faults. We would like to show analytically that such is the case. Clearly, for such analytical work, we should have a program behavior model. We will define a sequential-random model of program behavior for this purpose. Before that, we will define a demand prepaging algorithm based on the OBL algorithm of Joseph. We will follow the approach of King [KING71] to analyze the behavior of this algorithm.

DPOBLLRU = $(\bigvee, Q, N, S_0, q_0, g)$, where $\bigvee = \{S | S \subseteq N, |S| = c\}$ the set of memory states. S_0, the initial memory state, q_0, is the initial control state, Q is the set of control states $q = (j_1, j_2, \ldots, j_c)$ where $j_k \in N$ $(\forall k \in [1, c])$ and $j_i \neq j_\ell$ for $i \neq \ell$ (i.e., they are all distinct). g is the transition function, $g: \bigvee \times Q \times N \rightarrow \bigvee \times Q$.

$$g(S, q, x) = \begin{cases} (S, q') & \text{if } x \in S \\ (S'', q'') & \text{if } x \notin S,\ x \oplus 1 \in S \\ (S''', q''') & \text{if } x \notin S,\ x \oplus 1 \notin S. \end{cases}$$

Where $x \oplus 1$ denotes $x + 1 \bmod n$,

$q' = (j_1, \ldots, j_{k-1}, j_{k+1}, \ldots, j_c, x)$ where $j_k = x$.

$S'' =$ if $x \oplus 1 \neq j_1$ then $S - j_1 + x$;
else $S - j_2 + x$;

$q'' =$ if $x \oplus 1 \neq j_1$ then $(j_2, \ldots, j_c, x)$;
else $(j_1, j_3, \ldots, j_c, x)$;

$S''' = S - j_1 - j_2 + x + (x \oplus 1)$;

$q''' = (x \oplus 1, j_3, \ldots, j_c, x)$;

END DPOBLLRU;

From the algorithm, it is clear that given $q = (j_1, \ldots, j_c)$ we can conclude that $S = \{j_1, \ldots, j_c\}$. Therefore, a configuration (S, q) of the algorithm is completely specified by specifying q. Therefore, the set of configurations is the set Q. Our first objective is to seek the number of states in Q.

Definition: A state $q \in Q$ is said to have a property z (i.e., $z(q)$) if $\exists\ x \in N$ such that x and $x \oplus 1$ both occur in q.

Lemma 3.2:

$$z(q_0) \rightarrow \forall\ q \in Q,\ z(q).$$

Proof: The proof is by induction. In the transition function g of DPOBLLRU, assume that $z(q)$ and we will prove that $z(q')$, $z(q'')$ and $z(q''')$. Observe that the property z is invariant under a permutation of its argument. Therefore, $z(q) \Rightarrow z(q')$ since q' is a permutation of q.

Since $x \oplus 1 \in S$ and from the definition of S'', $x \oplus 1 \in S''$. Also from the definition of S'', $x \in S''$ therfore $z(q'')$. From the definition of q''', we have $z(q''')$. The result of the lemma immediately follows. □

Lemma 3.3:

$$|Q| = {}^{n}P_{c} - \frac{(n-c)!\ n}{(n-2c)!(n-c)}$$

where ${}^{n}P_{c}$ denotes the c-permutations out of n objects.

Proof: Berge [BERG71, p. 31] has proved that the number of subsets of size c (from the set N of n elements), which do not contain either two consecutive integers or both 1 and n simultaneously, is given by

$$f^*(n,c) = \binom{n-c}{c} \frac{n}{n-c} .$$

From Lemma 3.2 it follows that $|Q| = {}^{n}P_{c} - c!f^*(n,c)$.

Substituting for $f^*(n,c)$, we get the required result.

It has been mentioned that the set of configurations (Q) of DPOBLLRU form a Markov Chain. Assume that the transition probability matrix $P = (p_{q,q'})$ for the Markov Chain is given.

Lemma 3.4:

The Markov Chain of configurations of DPOBLLRU is irreducible provided $Pr[r_t=i] \neq 0$, $\forall\, i \in N$.

Proof: It is sufficient to show that $\forall\, q \in Q$, $\forall\, q' \in Q$, $\exists\, k > 0$, such that $p^{(k)}_{q',q} > 0$ where $p^{(k)}_{q',q}$ represents the probability of a k-step transition from state q' to state q. Starting with an arbitrary state $q' \in Q$ we will show a finite sequence of transitions to an arbitrary state $q \in Q$ and thus we will prove the lemma.

Let $q = (i_1, \ldots, i_c)$ and

$q' = (j_1, \ldots, j_c)$ and $q \neq q'$.

By applying Lemma 3.2 to q, we get, $\exists\, i \in N$ such that both i and $i \oplus 1$ occur in q. Let $j_\ell = i$, $1 \leq \ell \leq c$, and $i_s = i \oplus 1$, $1 \leq s \leq c$, $s \neq \ell$. Without loss of generality, assume that $s > \ell$. We present a sequence

of states starting from q' and leading to q and strictly following the transition function g. Clearly, there is an associated reference string, ω_1, such that $g^*(q', \omega_1) = q$. Note here that g^* is an extension of g and that we have omitted the memory state S, since it is completely specified by the control state. Now, since we give a transition from q' to q in a finite number of steps, ω_1 is a finite string. And since by our assumption $p(\omega_1) > 0$, we get the required result. [Note that in the following, an asterisk means unspecified or irrelevant page number.]

$$
\begin{aligned}
q' \to q_1 &= (*, \ldots\ldots\ldots *, j_1) && x = j_1 \\
q_1 \to q_2 &= (*, \ldots\ldots\ldots *, j_1, j_2) && x = j_2 \\
&\vdots && \vdots \\
q_{\ell-2} \to q_{\ell-1} &= (*, \ldots *, j_1, j_2, \ldots j_{\ell-1}) && x = j_{\ell-1} \\
q_{\ell-1} \to q_{\ell} &= (*, \ldots *, j_1, j_2, \ldots j_{\ell-1}, j_{\ell+1}) && x = j_{\ell+1} \\
&\vdots && \vdots \\
q_{s-3} \to q_{s-2} &= (*, \ldots *, j_1, \ldots, j_{\ell-1}, j_{\ell+1}, \ldots j_{s-1}) && x = j_{s-1} \\
q_{s-2} \to q_{s-1} &= (*, \ldots *, j_1, \ldots j_{\ell-1}, j_{\ell+1} \ldots j_{s-1}, j_{s+1}) && x = j_{s+1} \\
&\vdots && \vdots \\
\to q_{c-2} &= (*, *, j_1 \ldots, j_{\ell-1}, j_{\ell+1}, \ldots j_{s-1}, j_{s+1}, \ldots j_c) && x = j_c \\
q_{c-2} \to q_{c-1} &= (j_s, j_1, \ldots, j_{\ell-1}, j_{\ell+1}, \ldots j_{s-1}, j_{s+1}, \ldots j_c, j_\ell) && x = j_\ell .
\end{aligned}
$$

Thus with a reference string,

$$\omega_2 = j_1, \ \ldots, \ j_{\ell-1}, \ j_{\ell+1}, \ \ldots, \ j_{s-1}, \ j_{s+1}, \ \ldots, \ j_c,$$

we have reached the state q_{c-1}. Now with the following reference string,

$$\omega_3 = j_1, \ \ldots, \ j_{\ell}, \ j_{\ell+1}, \ \ldots, \ j_{s}, \ j_{s+1}, \ \ldots, \ j_c,$$

we will reach the state q. Note that all the intermediate states, while going from q_{c-1} to q, are permutations of q. Thus $\omega_1 = \omega_2\ \omega_3$ is the required reference string. Clearly, ω_1 is finite and therefore we have the required result. □

We define the sequential-random model of program behavior as follows:

$Pr[r_{t+1}=j \mid r_t=l] = p_{ij}$ is given by:

$$p_{ij} = \begin{cases} p_1 & \text{if } i = j, \\ p_2 & \text{if } j = i \oplus 1, \\ p_3 & \text{otherwise} \end{cases}$$

We require that $p_1 \geq p_2 > p_3 > 0$ and $p_1 + p_2 + (n-2)\ p_3 = 1$. We note that this model is closer to the behavior of real programs as compared to the independent reference model.

Conjecture 3.1:

$m(DPOBLLRU, c) \leq m(LRU, c)$ for the sequential-random model of program behavior. Where m(A, c) indicates the long term average page fault probability working with algorithm A and with c page frames of MM.

Unfortunately, we have not been able to prove this theorem. However, we can prove that $m(DPOBLLRU, 2) \leq m(LRU, 2)$.

3.3 PWS Algorithm

We assume that the program's reference string is fully known in advance. With this assumption, we will propose a new paging algorithm, PWS which will incur zero page faults, on the average. We will also derive the expressions for page pull probability and the average number

of memory page frames required and compare the results with WS algorithm of Denning [DENN68a].

Assume that $\omega = r_1, \ldots, r_L$ is the given reference string. Define the 'futuristic' working-set $W^*(t, \mathcal{T})$ by:

$$W^*(t, \mathcal{T}) = \{x \in N \mid \exists\, t',\ t \leq t' < t + \mathcal{T},\ r_{t'} = x\}.$$

In other words, W^* is the contents of a window of size $\mathcal{T}$, looking forward. The purpose of algorithm PWS is to keep W^* in MM. At any time t, if $r_{t+\mathcal{T}} \in W^*(t, \mathcal{T})$ then nothing needs to be done; otherwise, we issue instructions to fetch this page and thereby prepage it. We also determine whether $r_{t-1} \in W^*(t, \mathcal{T})$ and if not, we either push it from MM or declare it dead.

If it is assumed that $\mathcal{T} \geq T$, where T is the page-fetch time, then since a fetch instruction for a page is issued $\mathcal{T}$ time units in advance of its use, we will have no page faults whatsoever. In other words, we have $m(PWS, \mathcal{T}) = 0$ if $\mathcal{T} \geq T$.

We will now study properties of this model. From the above definitions the following properties can be proved. (We denote $|W^*(t,\mathcal{T})|$ by $\omega^*(t, \mathcal{T})$.)

P1: a) $W^*(t, 0) = \emptyset$ which implies $\omega^*(t, 0) = 0$

b) $W^*(L, \mathcal{T}) = \{r_L\}$, $\forall\, \mathcal{T} > 0$.

c) $\omega^*(t, \mathcal{T})$ is a monotonically non-decreasing function of $\mathcal{T}$.

d) $\omega^*(t, \mathcal{T})$ is concave downwards. To see this, note that,

$$W^*(t, 2\mathcal{T}) = W^*(t, \mathcal{T}) \cup W^*(t+\mathcal{T}, \mathcal{T}).$$

Therefore, $$\omega^*(t, 2\mathcal{T}) \leq \omega^*(t, \mathcal{T}) + \omega^*(t+\mathcal{T}, \mathcal{T}),$$

since the working sets may overlap.

Assuming statistical regularity, $\omega^*(t+\tau, \tau)$ behaves, on the average, like $\omega^*(t, \tau)$

therefore, $\omega^*(t, 2\tau) \leq 2\omega^*(t, \tau)$.

Hence we have the concavity property.

e) If $t < L - \tau$ then $W^*(t, \tau) = \{x \in N \mid x = r_{t'},\ t \leq t' \leq L\}$.

Let us denote

$$s_k^*(\tau) = \frac{1}{k} \sum_{t=1}^{k} \omega^*(t, \tau).$$

And let the average working set size $s^*(\tau) = s_L^*(\tau)$. The page pulling rate $m^*(\tau)$ measures the number of pages per unit time returning to the working set (note that, $m^*(\tau)$ will be an upper bound, since a page which has left $W^*(t, \tau)$ may still be in MM).

$$\text{Let } \Delta(t, \tau) = \left.\begin{array}{ll} 1 & \text{if } r_{t+\tau} \notin W^*(t, \tau) \\ 0 & \text{otherwise.} \end{array}\right\} \quad (t \geq 1)$$

Note that, $\Delta(t, \tau) = 0 \quad \forall\ t > L - \tau$. Let $m_0^* = \Delta(0, \tau) = \{r_1, \ldots, r_\tau\}$. The page pulling rate is given by:

$$m^*(\tau) = \lim_{k \to L} \left(\frac{1}{k} \left[\sum_{t=1}^{k-1} \Delta(t, \tau) + m_0^* \right] \right).$$

The interference distribution $F_i(x)$ and the interference density function $f_i(x)$ as well as the relative frequency of reference λ_i have the same meaning as in Denning's work which is outlined in Chapter 2. Let the mean overall reference interval $\bar{x} = \sum_{i=1}^{n} \lambda_i \bar{x}_i$, $\bar{x}_i = \sum_{x \geq 0} x\, f_i(x)$. The page i is called 'recurrent' if $\lambda_i \neq 0$ and γ_L denotes the number of recurrent pages in N. If $\lambda_i \neq 0$ then $\bar{x}_i = 1/\lambda_i$.

P2: $s^*(\mathcal{T})$ is nondecreasing in $\mathcal{T}$ and is bounded above and below:

$$1 = s^*(1) \leq s^*(\mathcal{T}) \leq s^*(\mathcal{T}+1) \leq \min.\{n, \mathcal{T}+1\}.$$

Proof: $W^*(t, \mathcal{T}) \subseteq W^*(t, \mathcal{T}+1) \Rightarrow \omega^*(t, \mathcal{T}) \leq \omega^*(t, \mathcal{T}+1)$.

$\Rightarrow s^*(\mathcal{T}) \leq s^*(\mathcal{T}+1)$. Also,

$\omega^*(t, 1) = 1 \Rightarrow s^*(1) = 1$.

$\omega^*(t, \mathcal{T}+1) \leq \min.\{n, \mathcal{T}+1\} \Rightarrow s^*(\mathcal{T}+1) \leq \min.\{n, \mathcal{T}+1\}$.

P3: $s^*(\mathcal{T}+1) - s^*(\mathcal{T}) = m^*(\mathcal{T}) - \frac{m_0^*}{L}$.

Note that $\frac{m_0^*}{L}$ is a very small number compared to $m^*(\mathcal{T})$ and can be ignored for most purposes, particularly if L is very large.

Proof:

$$W^*(t, \mathcal{T}+1) - W^*(t, \mathcal{T}) \subseteq \{r_{t+\mathcal{T}}\}.$$

Therefore, $$\omega^*(t, \mathcal{T}+1) - \omega^*(t, \mathcal{T}) = \Delta(t, \mathcal{T}).$$

Therefore, $$\lim_{k\to L} \left[\frac{1}{k} \sum_{t=1}^{k} \omega^*(t, \mathcal{T}+1) - \frac{1}{k} \sum_{t=1}^{k} \omega^*(t, \mathcal{T})\right]$$

$$= \lim_{k\to L} \frac{1}{k} \sum_{t=1}^{k} \Delta(t, \mathcal{T}).$$

Therefore, $$s^*(\mathcal{T}+1) - s^*(\mathcal{T}) = \lim_{k\to L} \left[\frac{1}{k} \sum_{t=1}^{k-1} \Delta(t, \mathcal{T}) + m_0^*\right]$$

$$+ \frac{\Delta(L, \mathcal{T})}{L} - \frac{m_0^*}{L}.$$

$$= m^*(\mathcal{T}) - \frac{m_0^*}{L}$$

$$\simeq m^*(\mathcal{T}). \qquad \square$$

P4: $\frac{\gamma_L}{L} \leq m^*(\tau+1) \leq m^*(\tau) \leq m^*(0) = 1.$

which states that $m^*(\ )$ is a nonincreasing function of τ and is bounded above and below. This follows immediately from P2 and P3.

P5: $m^*(\tau) = 1 - F(\tau) + \frac{m_0^*}{L}.$

which states that $m^*(\tau)$ can be regarded as the probability that $x > \tau$.

Proof: Define the binary variable $\beta_i(t, x) = 1$ iff $r_{t+\tau} = i$, $\exists\, u < t + \tau$ such that $r_u = i$ and $t + \tau - u \leq x$. Define $n_i(k)$ as the number of references to page i in $r_1, \ldots, r_k$. The definition of $F(x)$ can be expressed as:

$$F(x) = \sum_{i=1}^{n} \lambda_i F_i(x) = \sum_{i=1}^{n} \operatorname*{Lim}_{k\to L} \frac{n_i(k)}{k} \cdot \frac{1}{n_i(k)-1} \sum_{t=1}^{k-\tau} \beta_i(t, x)$$

$$= \operatorname*{Lim}_{k\to L} \frac{1}{k} \sum_{t=1}^{k-\tau} \sum_{i=1}^{n} \beta_i(t, x) \qquad \text{since } n_i(k)-1 \simeq n_i(k)$$

Define $\beta(t, x) = \sum_{i=1}^{n} \beta_i(t, x)$ and observe that $\beta(t, x) = 1 - \Delta(t, x)$.

Therefore, $1 - F(x) = \operatorname*{Lim}_{k\to L} \frac{1}{k} \sum_{t=1}^{k-\tau} \Delta(t, x)$

$$= m^*(x) - m_0^*/L.$$

This proves P5.

P6: $m^*(\tau+1) - m^*(\tau) = - f(\tau+1).$

This follows immediately from P5.

P7: $s^*(\tau) = \sum_{z=0}^{\tau-1} m^*(z) = \sum_{z=0}^{\tau-1} (1 - F(z) + \frac{m_0^*}{L})$

$$= \sum_{z=0}^{\tau-1} \sum_{y<z} f(y) + m^*_0 \tau/L.$$

This follows immediately from P5 and P6.

Since $m_0^* \tau/L$ is very small for large values of L (i.e., for long reference strings), $s^*(\tau) \simeq \sum_{z=0}^{\tau-1} \cdot \sum_{y<z} f(y)$.

Theorem 3.8:

a) $m(PWS, \tau) = 0$ if $\tau \geq T$,

b) $m^*(PWS, \tau) \simeq m(WS, \tau)$,

c) $s^*(PWS, \tau) \simeq s(WS, \tau)$,

d) $ST(PWS, \tau) \ll ST(WS, \tau)$.

Proof: Part a of the theorem has been already proved when we introduced the algorithm PWS. Part b follows from the property P5 of PWS and the property P4 of WS, given in section 2.4. Part c follows from the property P7 of PWS and the property P6 of WS, given in section 2.4. Finally, the last part can be shown as follows:

$$ST(PWS, \tau) = \Sigma\, \bar{\omega}^*(\tau) * (\delta t^*)$$

where δt^* = average execution burst plus the average page wait time plus the average ready queue delay.

$\bar{\omega}^*(\tau)$ is the average working set size during the interval δt^*.

Similarly, $ST(WS, \tau) = \Sigma\, \bar{\omega}(\tau) * t$.

Now since approximately $\bar{\omega}^*(\tau) \simeq \bar{\omega}(\tau)$ and $\delta t^* \ll \delta t$ since we have no page faults in PWS, $\delta t^* \simeq$ average execution burst. Therefore, we have the required result. □

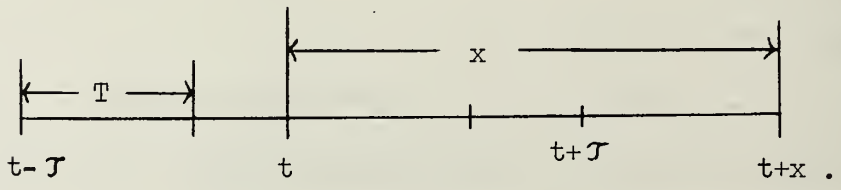
Implication of Theorem 3.8 is that we have defined a paging algorithm with zero page faults which pulls the same number of pages as the WS algorithm and which requires the same average working set size

as WS. We have also proved that the memory utilization (ST product) of PWS is highly superior to that of WS. The main drawback, however, is that PWS is unrealizable. If partial information is available about future working sets then we may be able to utilize it to improve performance over WS and at least approach the performance of PWS. We feel that further investigations should be made in this direction. We will add some comments to this topic in Chapter 6.

We have ignored several points in the above discussion of PWS which we now discuss. Define the residency of a page in MM as the fraction of time it is potentially available in MM. We assume that a page in the working set is never replaced. Once a page has entered $W^*(t, \tau)$, it will remain in MM for at least τ time units. Let x be the interreference interval for the page i. We have:

1) If $x \leq \tau$, the page residency is 100%.

2) If $\tau < x$ then the following diagram gives us the page residency.



If during the last residency, the page was updated, then the above diagram requires that $t + x - \mathcal{T} \geq t + T$ in order for the page to be released at time t. Which means that if $x \geq T + \mathcal{T}$ then only the page i can leave memory. Otherwise (i.e., $\mathcal{T} < x < T + \mathcal{T}$) then if the page was allowed to leave memory and called again after x time units then the contents of page i will be the old contents of page i (i.e., before the update during the last residency). Therefore, in such a case, the page must be kept in MM and it will always be resident. This means that more than $\omega^*(t, \mathcal{T})$ pages will occupy MM. To implement this in practice, we assume that the reference string $\omega = r_1, r_2, \ldots, r_L$ is such that $r_t = (y, k)$ where $y \in N$ is the current page referenced and k is the time to next reference to y. The value of k will be zero if page y if not referenced again. The modified PWS, therefore, can be written as follows:

MPWS:

(assume $\mathcal{T} \geq T$)

$\forall\, t \geq 1$ let $r_{t-1} = (y, k)$ and $r_{t+\mathcal{T}} = (y', k')$.

1) If $k = 0$ or $k \geq T + \mathcal{T}$ then

FREE(y)

2) If $y' \notin W^*(t, \mathcal{T})$ then PREPAGE (y').

END MPWS;

4. IMPROVING LOCALITY OF ARRAY PROGRAMS

4.0 Introduction

Locality of programs can be improved in various ways, as noted in Chapter 2. By proper organization of data among the data pages, we can reduce the size of the localities as well as the probability of an interlocality transition. This has been termed the problem of data pagination. For matrix algorithms operating on large arrays, we need to consider various techniques for storing them, such as, row major order, column major order, packed row storage, packed column storage, or submatrix (block) storage. Several investigations have indicated that submatrix storage is preferable to other storage schemes [MCKE69, ROGE73]. We shall restrict our investigation to the submatrix type of organization. For large matrix problems, it should be clear that instruction paging is dominated by data paging; for this reason we ignore instruction paging.

With respect to improving locality, it is possible, in certain instances, to resequence some of the operations in a program so that we operate on the data that is already resident in MM rather than execute unnecessary paging. This amounts to a logical reorganization of the code as against pagination, which does a physical reorganization. In this chapter, we will include a discussion of some of these methods of logical reorganization for matrix programs. We will assume that these

modifications are done by the programmer. The problem of automating some of these methods will be considered in Chapter 6.

Since we are interested in improving locality, we are also interested in measuring these improvements. We feel that the average working set size $s(\mathcal{T})$ as a function of $\mathcal{T}$ is a very good indication of the locality of a program. We also measure the number of page faults using LRU, WS and MIN paging algorithms.

We will present three techniques for the improvement of locality of matrix programs. We will illustrate their use by applying them on several common matrix algorithms. We will show that an order of magnitude improvement in locality can be obtained by these methods.

We now consider the notation for storing matrices in VA space. Let A be a rectangular matrix of size $(n_1 \times n_2)$. The matrix will be divided into square submatrices of order m as shown in Figure 4.1. If we assume that p, the page size, is a perfect square and that $m^2 = p$ then each of the submatrices can be stored in one page. Note that there will be some submatrices on the right and the bottom borders which will not be full. We will assume that n_1 and n_2 are both multiples of m and therefore, such fragmentation will not occur. If the matrix A is a square matrix then $n_1 = n_2 = n$. We define the integers N, N_1 and N_2 through the equations $n = N \cdot m$, $n_1 = N_1 \cdot m$ and $n_2 = N_2 \cdot m$. Therefore, the number of data pages occupied by a $n_1 \times n_2$ matrix is $N_1 \cdot N_2$, by a $n \times n$ matrix is N^2 and by a triangular matrix of order n is $N(N+1)/2$.

4.1 Cholesky Decomposition

We start with a particular matrix algorithm, namely, the Cholesky factorization of a symmetric, positive definite matrix. We will try to

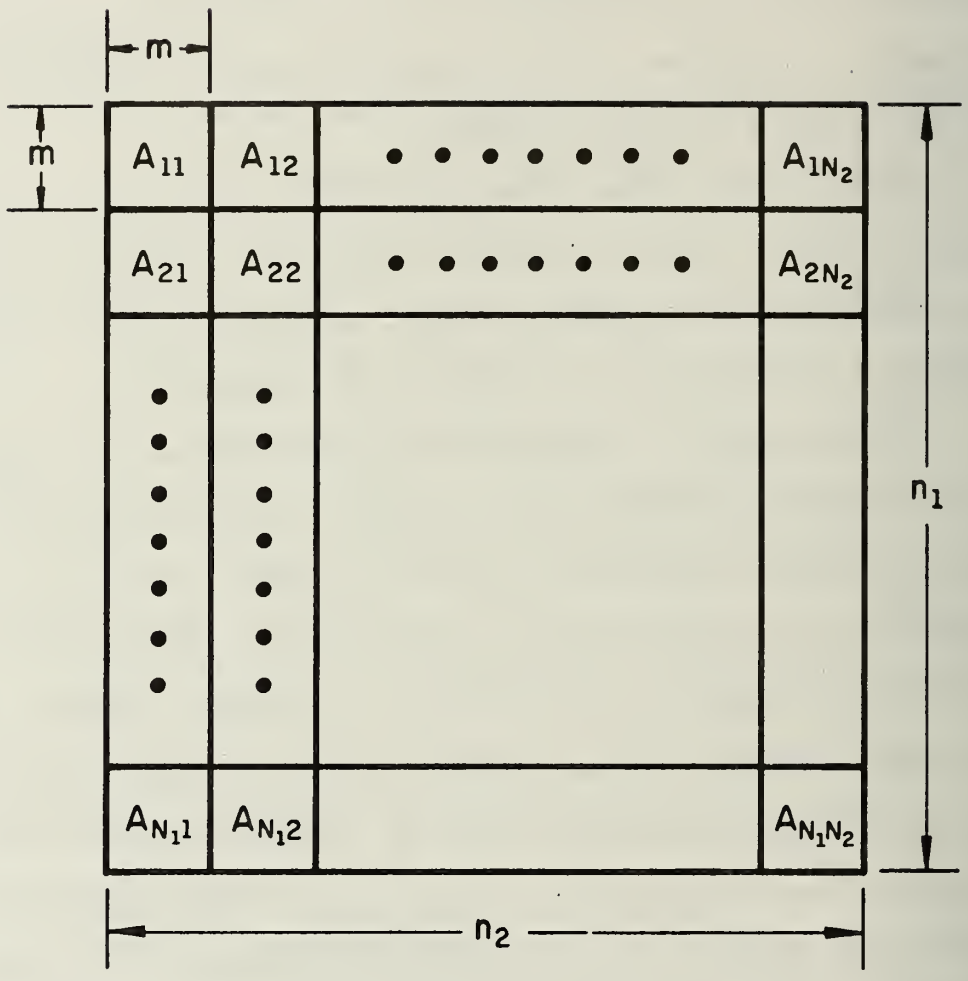


Figure 4.1. Submatrix Storage

improve the locality of this algorithm in various ways. This will, then, lead us to general methods of locality improvement. We will try these methods on some other typical matrix algorithms.

Cholesky decomposition factors a symmetric positive definite matrix A into the product G×G' where G is a lower triangular matrix and G' denotes the transpose of G [WILK71]. The algorithm is defined by the following relations:

$$\left.\begin{aligned} g_{kk} &= \mathrm{sqrt}\left(a_{kk} - \sum_{j=1}^{k-1} g_{ij}^{2}\right) \\ g_{ik} &= \left(a_{ik} - \sum_{j=1}^{k-1} g_{jk}\, g_{ik}\right)/g_{kk}, \qquad \text{for } 1 \le k < i \le n. \end{aligned}\right\} \quad \text{for } 1 \le k \le n$$

Where $A = [a_{ij}]$ is an n×n matrix, and $G = [g_{ij}]$ is a lower triangular matrix of order n. We recall that the basic algorithm in PL/I like language is as in Figure 4.2. Note that, we operate only on the lower triangular part of A and we overwrite G onto A. We have omitted all declarations, input, output and such other statements from the program. Note that, for our measurements, we will use a matrix (A) of order n = 24, a page size p = 16 ($\Rightarrow$ m = 4). We note that the first inner loop, 'DO J = 1 TO k - 1', scans the k^{th} row of the matrix A. This is done with each traversal of the major loop (with a different value of k). Since several (in fact, m) rows are stored across in a row of pages, it is clear that the locality will be improved if we alternately traverse the rows in opposite directions. We observe that these traversals are possible because the order of computation within the loop is immaterial. By appropriately reversing the other two loops, we will improve the locality in a similar way. The resulting program is labelled CDR, which is given in Figure 4.3.

```
CD:  /* CHOLESKY DECOMPOSITION */
DO k = 1 TO n;
 S = 0;
 DO J = 1 TO k - 1;
  S = S + A(k, J) * A(k, j);
 END;
 A(k, k) = SQRT(A(k, k) - S);
 DO I = 1 TO n - k;
  S = 0;
  DO J = 1 TO k - 1;
   S = S + A(k+I, J) * A(k, J);
  END;
  A(k+I, k) = (A(k+I, k) - S)/A(k, k);
 END;
END CD;
```

Figure 4.2. Cholesky Decomposition

```
CDR:
  DO k = 1 TO n;
   S = 0;
   IF MOD(k, 2) = 1 THEN
      DO; JLL = 1; JHL = k - 1; JSTEP = 1; END;
                    ELSE
      DO; JLL = k - 1; JHL = 1; JSTEP = -1; END;
   DO J = JLL TO JHL BY JSTEP;
    S = S + A(k, J) * A(k, J);
   END;
   A(k, k) = SQRT(A(k, k) - S);
   IF MOD(k, 2) = 1 THEN
      DO; IHL = n - k; ILL = 1; ISTEP = 1; END;
                   ELSE
      DO; IHL = 1; ILL = n - k; ISTEP = -1; END;
   DO I = ILL TO IHL BY ISTEP;
    IF MOD(I, 2) = 1 THEN
       DO; JLL = 1; JHL = k - 1; JSTEP = 1; END;
                     ELSE
       DO; JLL = k - 1; JHL = 1; JSTEP = -1; END;
    S = 0;
   DO J = JLL TO JHL BY JSTEP;
    S = S + A(k+I, J) * A(k, J);
   END;
   A(k+I, k) = (A(k+I, k) - S)/A(k, k);
END CDR;
```

Figure 4.3. Cholesky Decomposition with Reversal

In Figure 4.4, we have plotted the average working set size $s(\tau)$ as a function of the window width τ for CD and CDR. The improvement of locality due to the reversal technique is clearly shown.

The locality of a program, such as CD, may be improved more significantly if we extend the algebra of the language to include submatrix operations. We sketch the program using the OL/2 language [PHIL72]. The names of all OL/2 programs will be prefixed by the letter 'O'. In Figure 4.5, we give the program OCD.

The order of the element by element computation of $(C - M \times R')$ is unspecified in OCD. Since a matrix-vector multiplication is specified by $M \times R'$, we have the choice of implementing the multiplication by a submatrix multiplication and thereby improve the locality. We will briefly explain what we mean by submatrix multiplication. Assume we are carrying out $Z = X * Y$. Where X, Y and Z are matrices of order n. We shall denote the (I, J) submatrix of the matrix X by the usual subscript notation, e.g., X_{IJ}. Multiplication of X and Y can, therefore, be written as follows:

$$Z_{IJ} = \sum_{k=1}^{N} X_{IK} * Y_{KJ} \qquad 1 \leq I, J \leq N.$$

Note that, the sum and product in the above equation denotes the matrix sum and matrix product for matrices of order m. We apply this submatrix operation to modify CD and obtain the results shown in Figure 4.7 for the program CDM in Figure 4.6.

We have plotted, in Figure 4.7, $s(\tau)$ vs. τ for CD and CDM. We note that the improvement obtained by this method is much more than that obtained by the method of loop reversal.

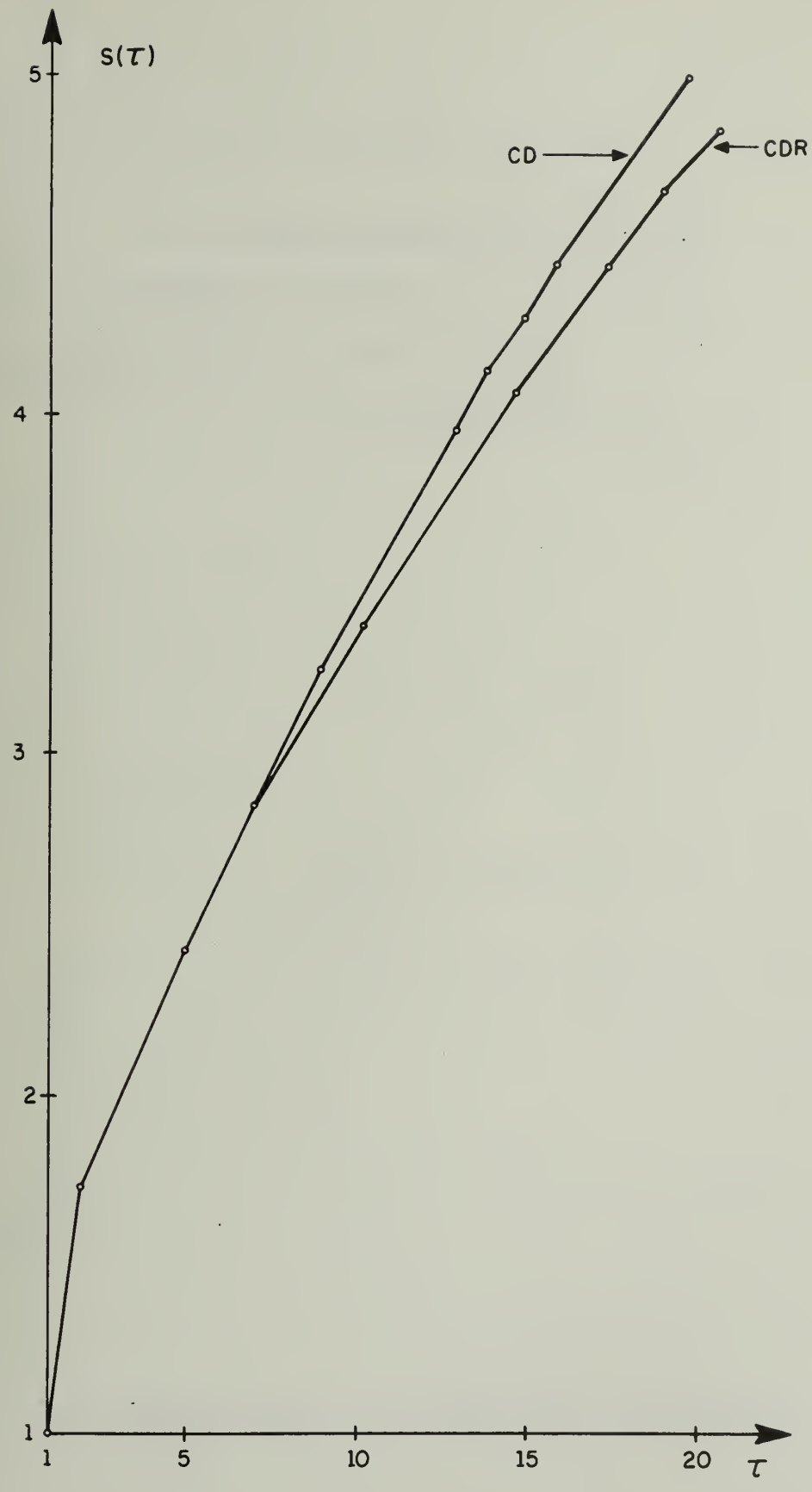


Figure 4.4. Localities of CD and CDR

```
OCD:  PROC(A, n);
 LET A BE A LOWER TRIANGULAR MATRIX OF
 ORDER (n);
 FOR K = 1, 2, ..., n; PARTITION A AFTER ROWS
  K - 1, K;
SET R = A <2,1> ROW VECTOR, D = A <2,2> SCALAR,
    M = A <3,1>           , C = A <3,2> COLUMN VECTOR;
    D = SQRT(D - (R', R'));
    C = (C - M × R')/D;
END OCD;
```

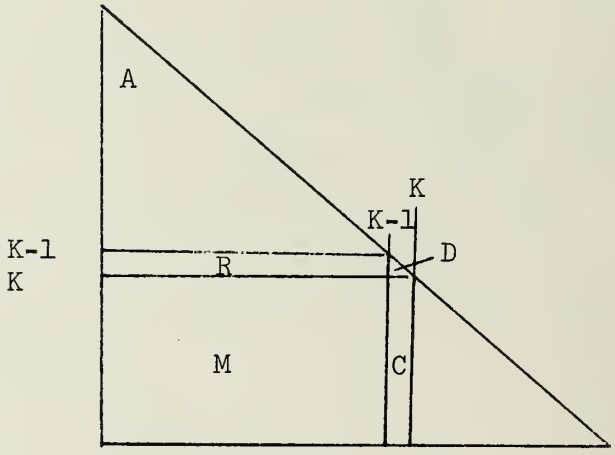


Figure 4.5. OL/2 Program for Cholesky Decomposition

```
CDM:
 FM = FLOAT(M);
 DO K = 1 TO n; K1FM = (K-1)/M; KCM = CEIL(K/FM);
  S = 0;
  DO J = 1 TO K - 1;
   S = S + A(K, J) * A(K, J);
  END;
  A(K, K) = SQRT(A(K, K) - S);
  DO J = 1 TO K1FM;
   DO I = K+1 TO KCM * M;
    S = 0;
    DO J1 = 1 TO M;
     S = S + A(I, (J-1) * M + J1) * A(K, (J-1) * M + J1);
    END;
    A(I, K) = A(I, K) - S;
   END;
  END;
   DO I = K+1 TO KCM * M;
    S = 0;
    DO J = K1FM * M + 1 TO K - 1;
     S = S + A(I, J) * A(K, J);
    END;
    A(I, K) = (A(I, K) - S)/A(K, K);
   END;
  DO I = KCM + 1 TO N;
   DO J = 1 TO K1FM;
    DO I1 = 1 TO M;
     S = 0;
     DO J1 = 1 TO M;
      S = S + A((I-1) * M + I1, (J-1) * M + J1) *
              A(K, (J-1) * M + J1);
     END;
     A((I-1) * M + I1, K) = A((I-1) * M + I1, K) - S;
    END;
   END;
   DO I1 = 1 TO M;
    S = 0;
    DO J = K1FM * M + 1 TO K - 1;
     S = S + A((I-1) * M + I1, J) * A(K, J);
    END;
    A((I-1) * M + I1, K) = (A((I-1) * M + I1, K) - S)/A(K, K);
END CDM;
```

Figure 4.6. Cholesky Decomposition with Submatrix Multiplication

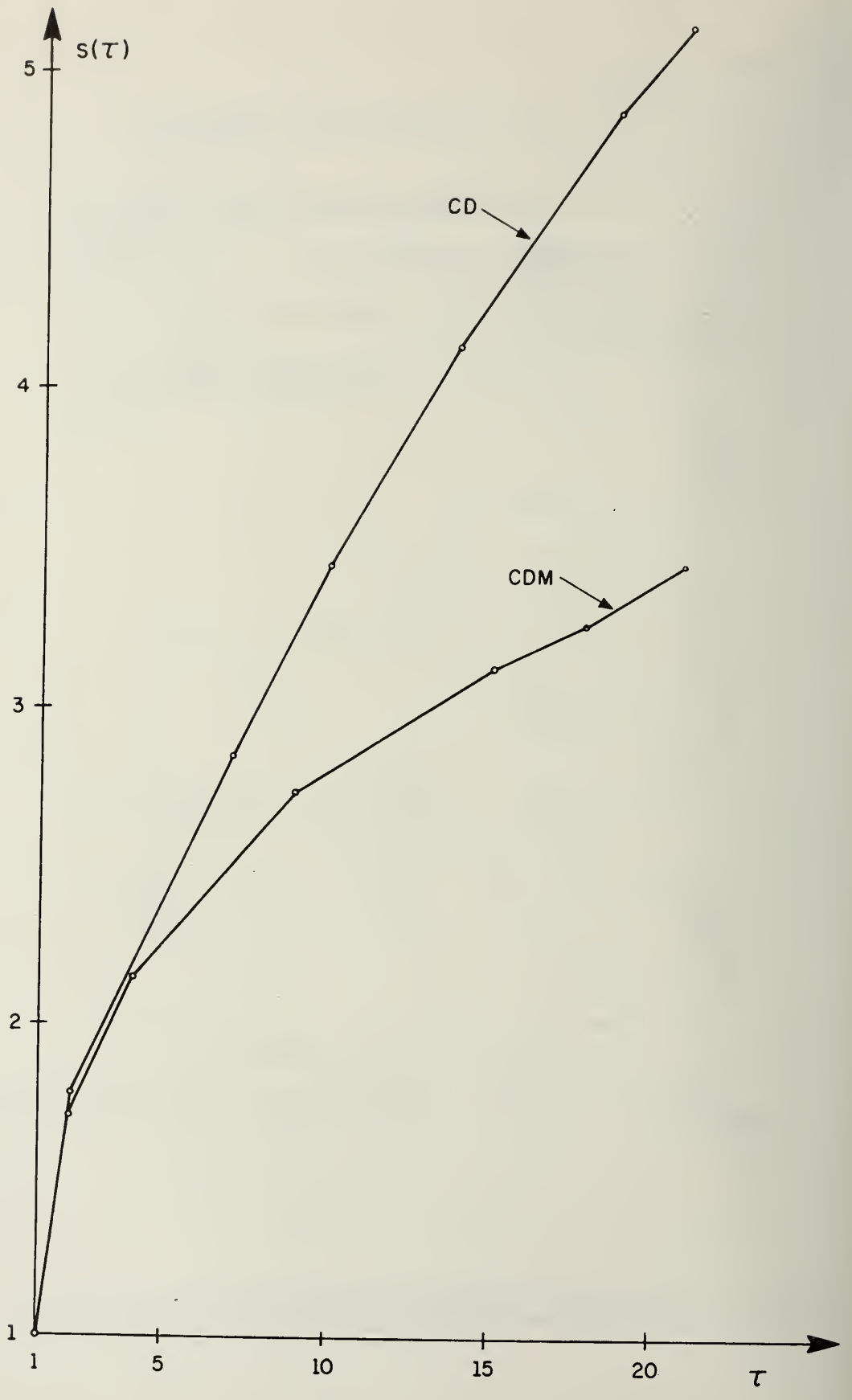


Figure 4.7. Localities of CD and CDM

We now turn to an even better technique for improving the locality. This requires the construction of a submatrix algorithm and we illustrate it for the Cholesky factorization [HOUS64].

$$\left.\begin{aligned} G_{II} &= \text{CHOLDEC}\left(A_{II} - \sum_{J=1}^{I-1} (G_{IJ} \times G'_{IJ})\right) \\ G_{JI} &= \left(A_{JI} - \sum_{K=1}^{I-1} (G_{JI} \times G'_{IK})\right) \times (G'_{II})^{-1} \\ &\qquad \text{for } I < J \leq N \end{aligned}\right\} \quad \text{for } 1 \leq I \leq N.$$

We now present the OL/2 program for this algorithm. A PL/I program, CDS, is given in Appendix C.

```
OCDS:  PROC;
  FOR K = 1, 2, ..., N; PARTITION A AFTER
   ROWS m * (K-1), m * K;
SET R = A <2,1>     ,     D = A <2,2>,
    C = A <3,2>     ,     M = A <3,1>;

    D = D - R × R'
    CALL OCD (D, m);
    C = (C - M × R') × (D')**-1;

END OCDS;
```

Note that, D is a lower triangular matrix of order m, R is a rectangular matrix of size $m \times m(k-1)$. Similarly, C is $(N-m)k \times m$ and M is $(N-m)\ k \times m(k-1)$ matrices respectively. To carry out the Cholesky factorization of D, only one page is involved, and therefore, any method can be used without changing the performance. Therefore, we use OCD for simplicity. Based on OCDS, we write CDS given in Appendix C. We carry out the multiplication M × R' by submatrices. We store the inverse of D' in the strictly upper triangular part of D. This way, we will not be able to store the diagonal elements of $(D')^{-1}$. Note that, this storage scheme is possible because the page containing the subarray D has

precisely m(m-1)/2 words vacant. Also note that, the diagonal elements of $(D')^{-1}$ are the reciprocals of the corresponding diagonal elements of D.

In Figure 4.8, we have plotted $s(\tau)$ vs. τ for CD and CDS. The large improvement in locality is quite obvious. We note that the reversal technique can be applied to both, CDM and CDS obtaining CDMR and CDSR respectively. Improvements thus gained are not significant enough to report. In Figure 4.9, we have plotted $s(\tau)$ vs. τ for CD, CDR, CDM and CDS. It can be easily seen that the versions of Cholesky factorization in order of increasingly better locality are CD, CDR, CDM and CDS.

In Figure 4.10, we have plotted the number of page fault, $\pi(\tau)$ vs. τ for CD, CDR, CDM and CDS using WS paging algorithm. In Figure 4.11 and Figure 4.12, we have plotted $\pi(c)$ vs. c (the page allotment) for CD, CDR, CDM and CDS using LRU and MIN paging algorithms respectively. It is clear from these figures that, improved locality implies a reduction in page faults.

4.2 Other Matrix Algorithms

From the example of Cholesky decomposition, we observe that there are three general methods of locality improvement of matrix algorithms. The first method is the method of loop reversal in which we change the direction of loop traversal each time we traverse a loop. We note that, depending on the computation within the loop, this may not always be possible. The second method is to carry out all matrix multiplications occurring within the algorithm as submatrix multiplications. The third method consists in using a submatrix algorithm from the beginning. We

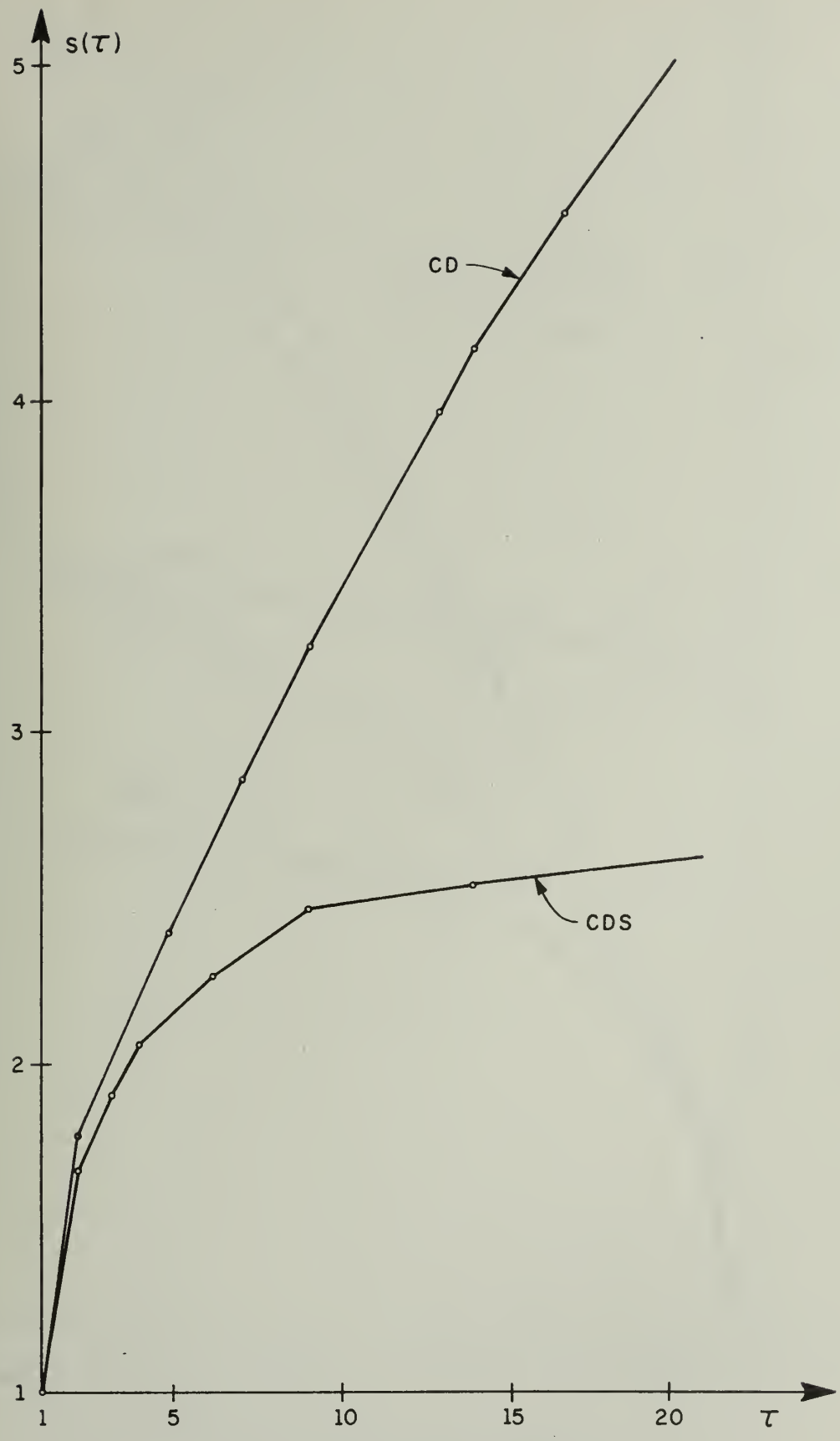


Figure 4.8. Localities of CD and CDS

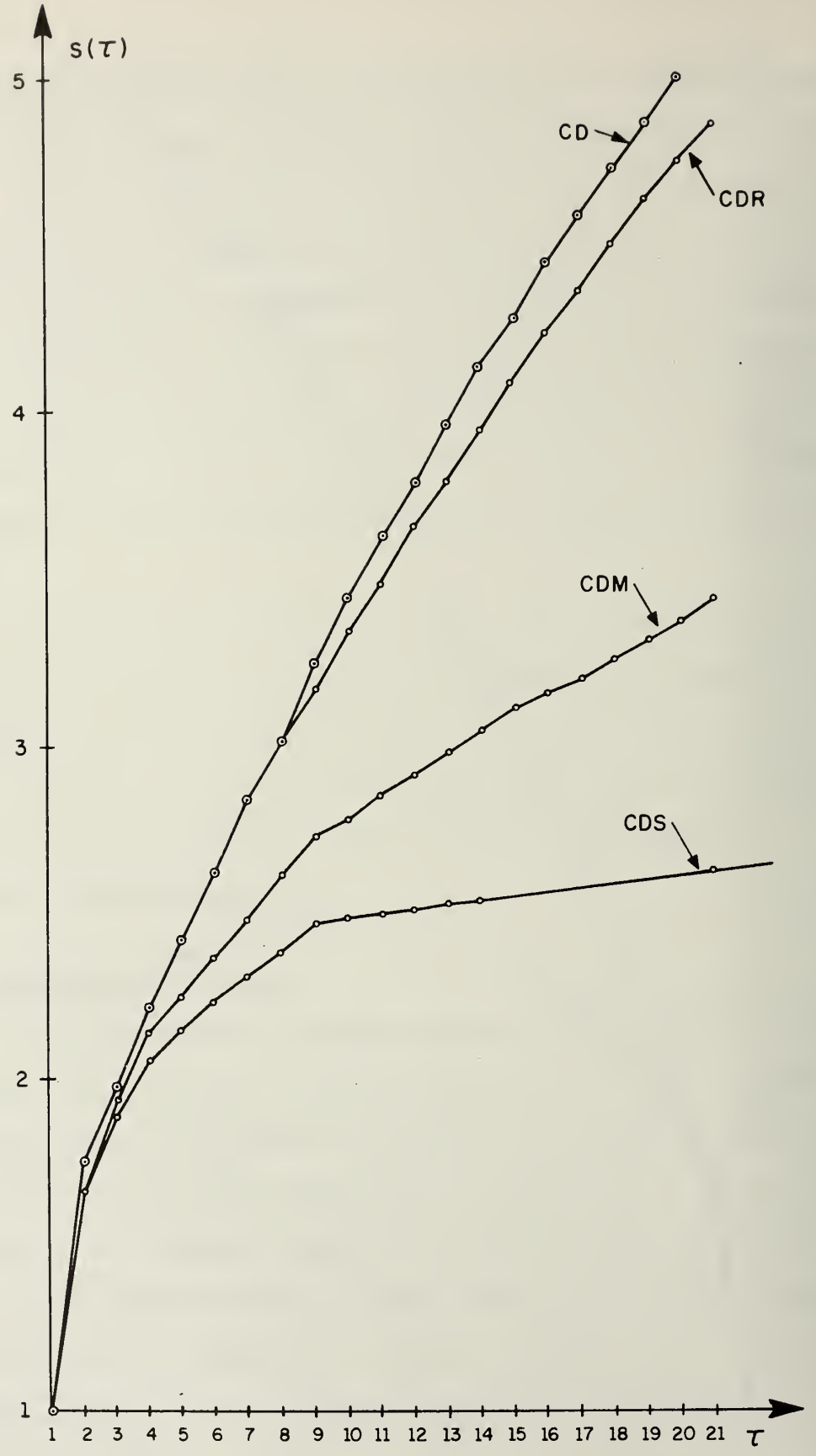


Figure 4.9. Localities of CD, CDR, CDM and CDS

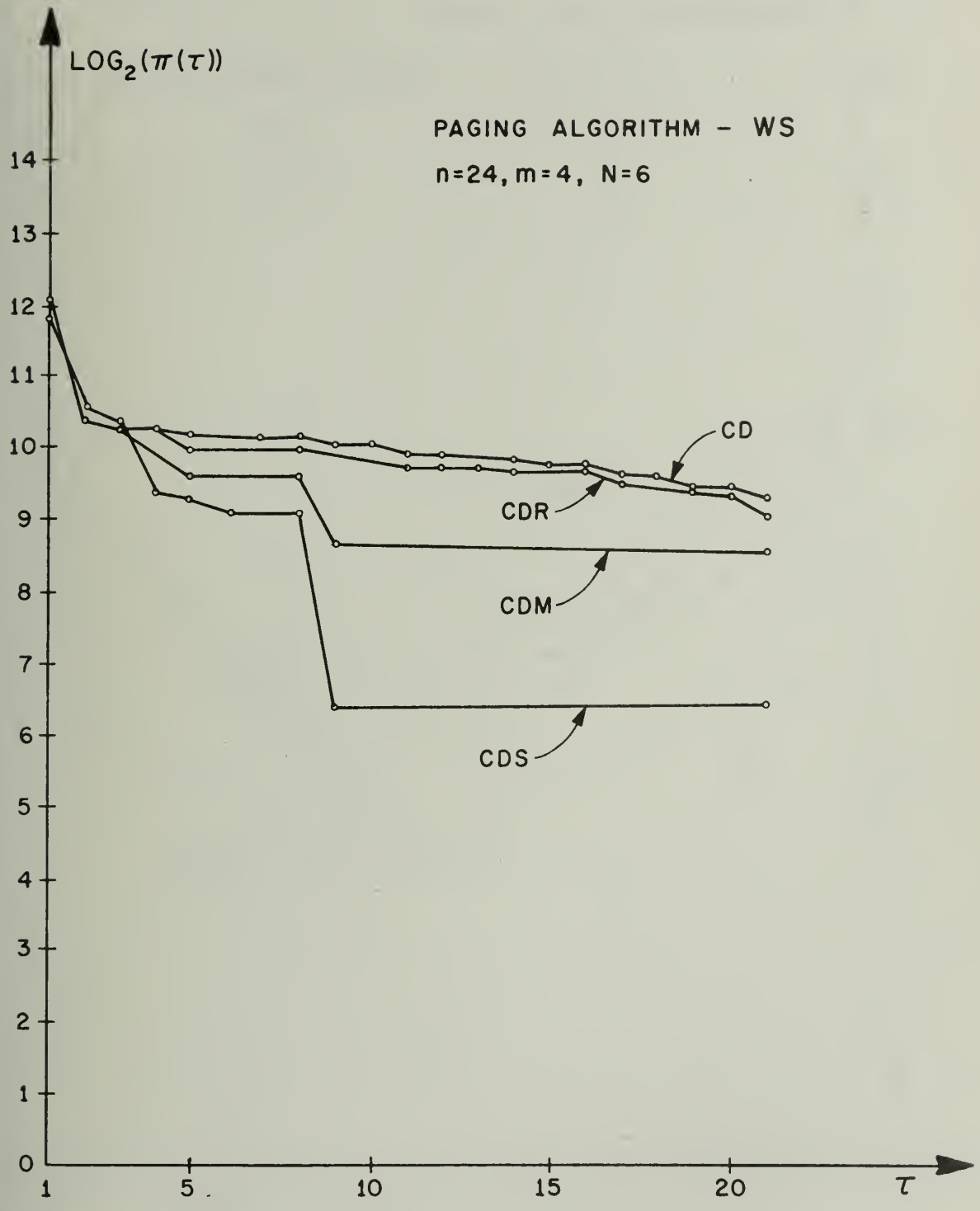


Figure 4.10. Page Faults Using WS Algorithm

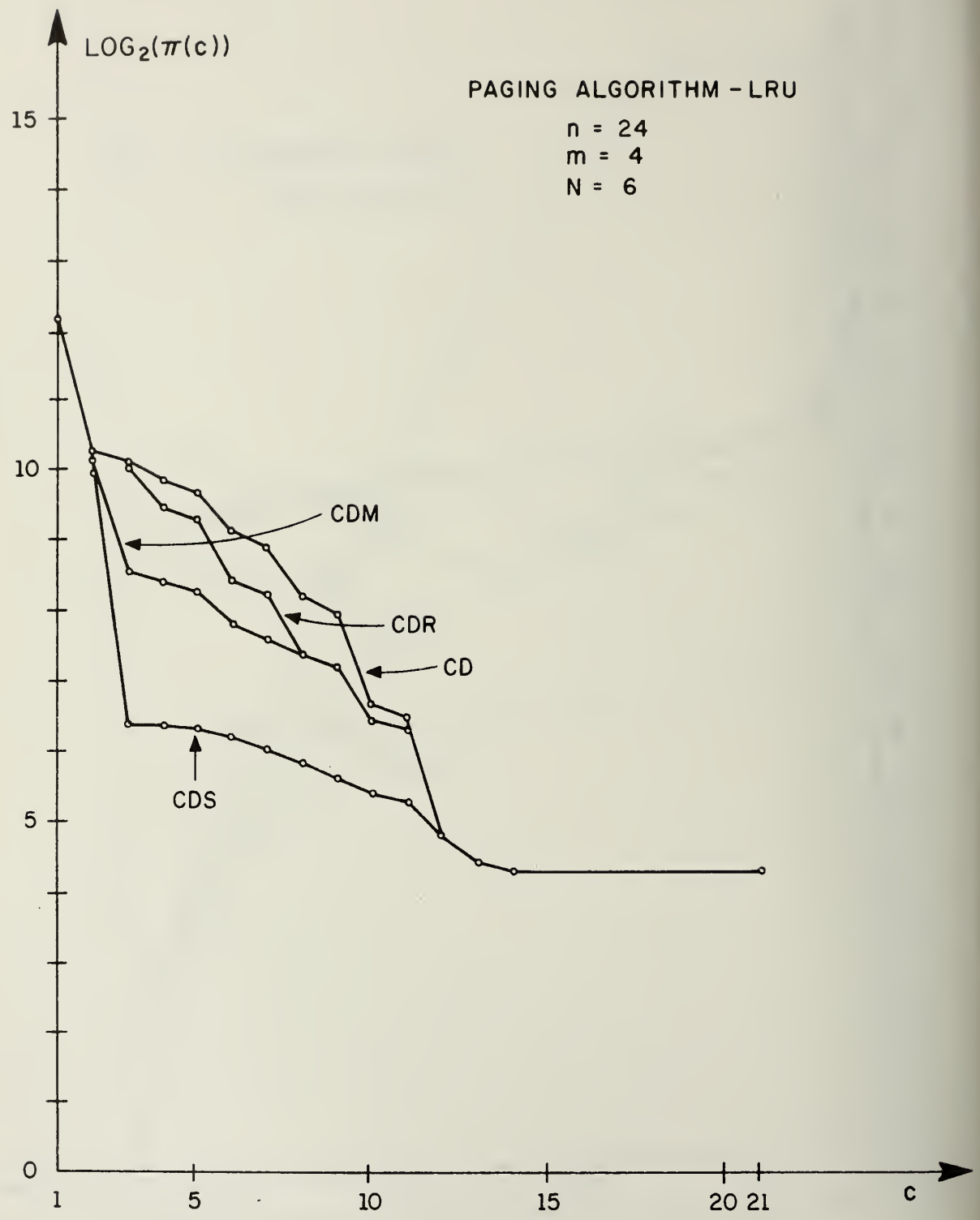


Figure 4.11. Page Faults Using LRU Algorithm

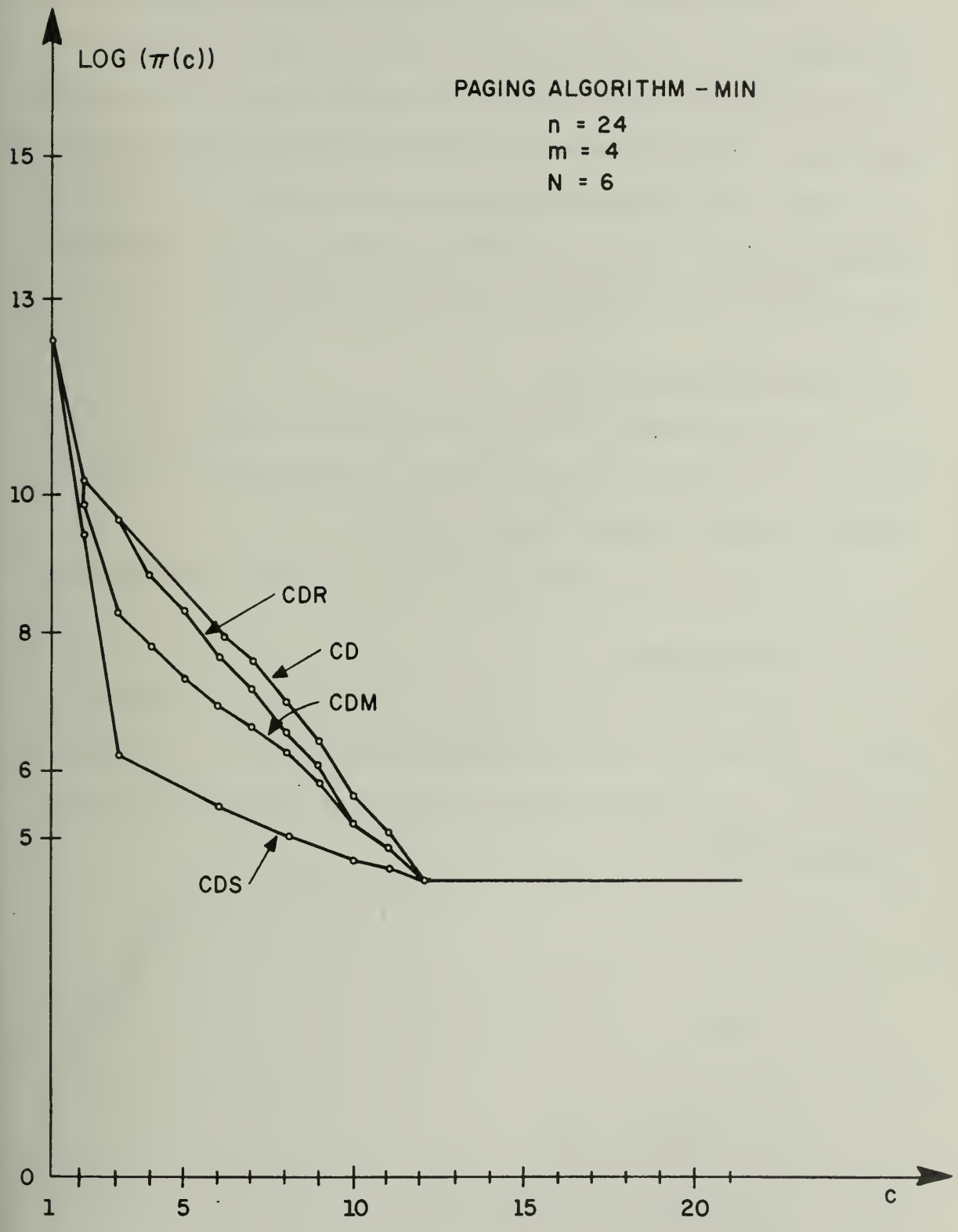


Figure 4.12. Page Faults Using MIN Algorithm

note that, for several matrix algorithms, their submatrix versions already exist in the literature. However, for eigenvalue problems, such submatrix algorithms do not exist and probably will not be found in the future. We believe that the three methods can be usefully applied to most matrix algorithms. We will consider several more algorithms in this chapter. We note that, the conclusions regarding the performance improvements by these methods hold for any matrix size and any page size, although we have considered only one matrix size and one page size.

4.2.1 Matrix Multiplication

Three versions of matrix multiplication we consider are: MM (basic matrix multiplication), MR (matrix multiplication with loop reversal, and MMS (submatrix multiplication).

In Figure 4.13, we plot $s(\mathcal{T})$ vs. $\mathcal{T}$ for these three programs.

4.2.2 LU Decomposition

Given a nonsingular matrix A, it can be uniquely decomposed into L × U where L is a lower triangular matrix and U is unit upper triangular. The method is known as Crout LU decomposition, and can be written as [ISAA66]:

```
for k = 1 to n do;
    for i = k to n do;
        l_ik = d_ik - Σ_{j=1}^{k-1} l_ij u_jk;
    end;
    for j = k+1 to n do;
        u_kj = (a_kj - Σ_{i=1}^{k-1} l_ki u_ij)/l_kk;
    end;
    u_kk = 1;
end;
```

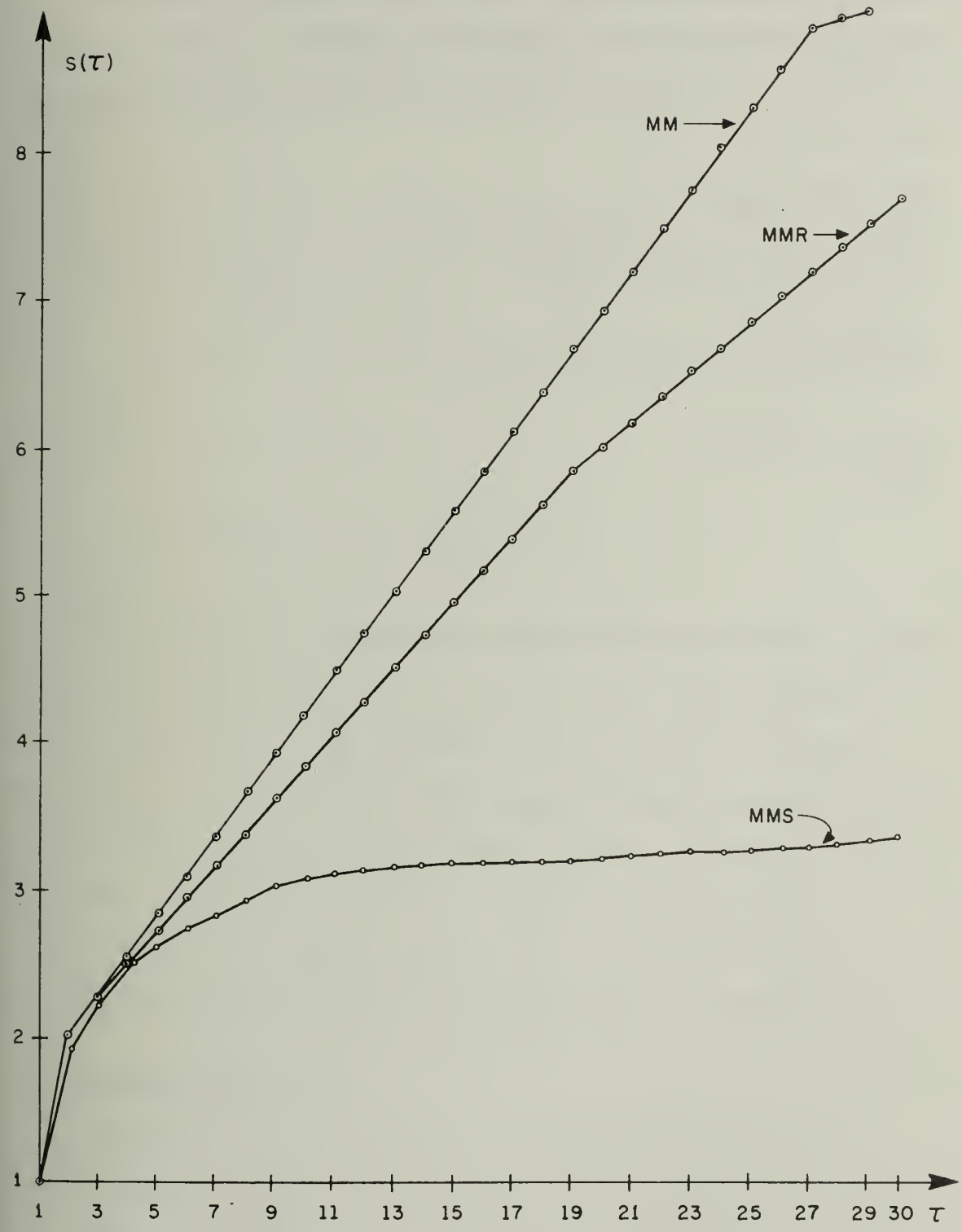
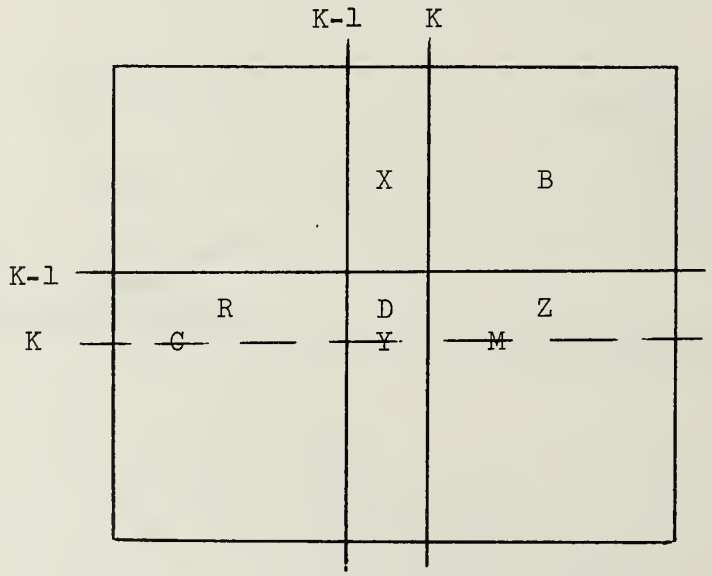


Figure 4.13. Localities of MM, MMR and MMS

A straightforward implementation of this will be called LU. Appropriate loop reversal turns it into LUR. To obtain LUM, we must identify matrix multiplications within the algorithm. Towards this end, we write the algorithm in OL/2 and call it OLU. Note that, we store L, U over A.

```
OLU:  PROC(A, n);
 LET A BE A MATRIX OF ORDER(n);
 FOR K = 1, 2, ..., n; PARRITION A AFTER ROW
 K-1 AND AFTER COLUMNS K-1 and K;
 SET C = A <2,1>, B = A <1,3>, X = A <1,2>, Y = A <2,2>,
     M = A <2,3>;
 PARTITION C, Y, M AFTER ROW 1;
 SET R = C <1>, D = Y <1>, Z = M <1>;

 Y = Y - C * X;
 Z = (Z - R * B)/D;
END OLU;
```



From OLU, we see that C * X is a matrix-vector product and R * B is a vector-matrix product. Coding these products as submatrix products (as in CDM) we obtain LUM (we omit the detailed coding here). Submatrix LU decomposition algorithm can be written as [ISAA66]

for K = 1 to N do;

$$(L_{KK}, U_{KK}) \leftarrow \text{LU\_DEC}(A_{KK} - \sum_{J=1}^{K-1} L_{KJ}\, U_{JK});$$

for I = K + 1 to N do;

$$L_{IK} \leftarrow (A_{IK} - \sum_{J=1}^{K-1} L_{IJ}\, U_{JK}) * (U_{KK})^{-1};$$

end;

for J = K + 1 to N do;

$$U_{KJ} \leftarrow (L_{KK})^{-1} * (A_{KJ} - \sum_{I=1}^{K-1} L_{KI}\, U_{IJ});$$

end K;

This can easily be programmed into OL/2 as well as PL/I, obtaining OLUS and LUS respectively. In both cases, we store L and U on the original storage of A. While programming LUS, we store the inverses of L_{KK} and U_{KK} on a scratch page (only one scratch page is needed). Once again, we do not give the detailed codes for these. In Figure 4.14, we have plotted $s(\mathcal{T})$ vs. $\mathcal{T}$ for LU, LUR, LUM and LUS. We have used $n = 24$ (= order of the matrix A), page size $p = 16$ (implies $m = 4$). Thus the number of pages occupied by A is $6 \times 6 = 36$ (in general, N^2).

4.2.3 Gaussian Elimination

Gaussian elimination reduces a nonsingular matrix A into an upper triangular matrix. Gaussian elimination with coefficient storage can be written as [MCKE69]:

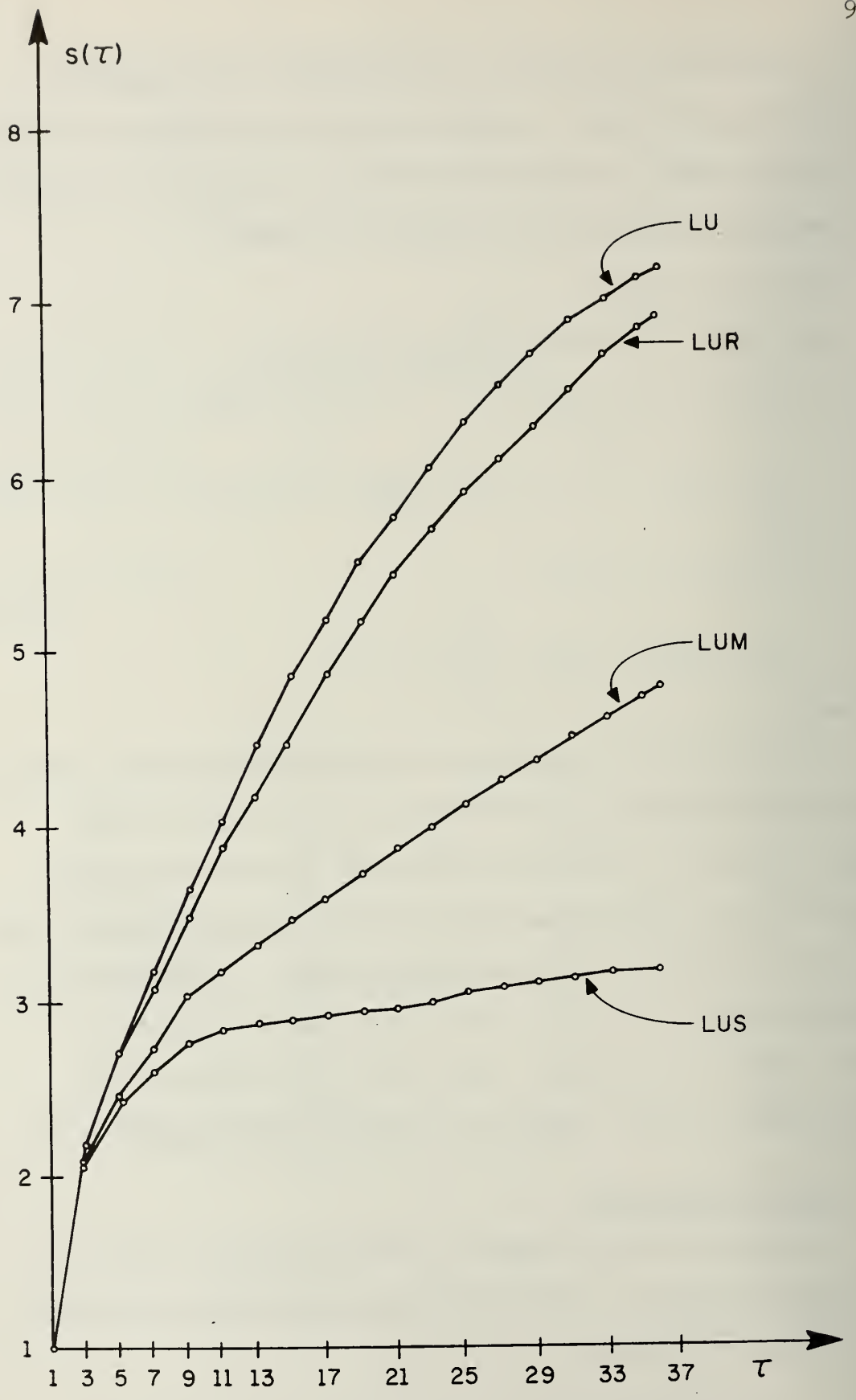
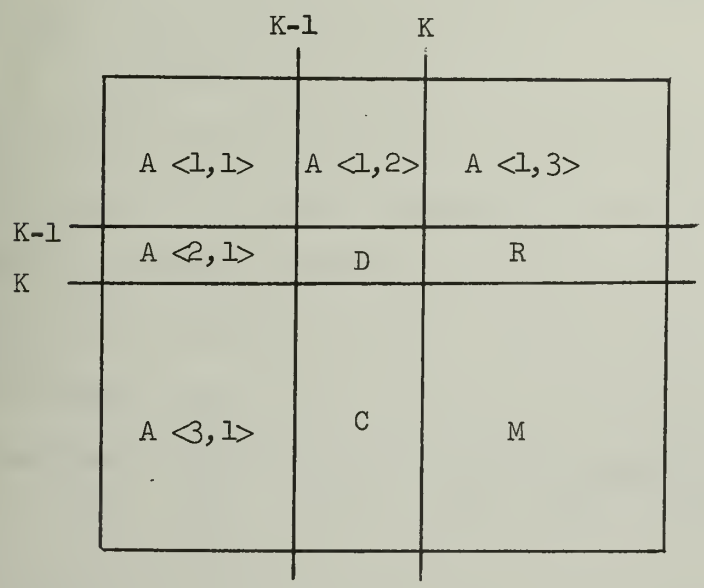


Figure 4.14. Localities of LU, LUR, LUM and LUS

```
    for k = 1 to n - 1 do;
       for i = k + 1 to n do;
         a_ik = a_ik/a_kk;
       end;
       for j = k + 1 to n do;
            for i = k + 1 to n do;
              a_ij = a_ij - a_ik * a_kj ;
       end;
    end;
 end;
```

A straightforward implementation of this algorithm is called GOS (we omit the coding here). From GOS with loop reversal, we easily obtain GOSR (code omitted here). To see the matrix multiplications within GOS, we write it in OL/2 and call it OGOS.

```
OGOS:  PROC(A, n);
 LET A BE A MATRIX OF ORDER(n);
 FOR K = 1, 2, ..., n-1; PARTITION A AFTER
 ROWS K-1, K AND AFTER COLUMNS K-1, K;
SET D = A <2,2> SCALAR, R = A <2,3> ROW VECTOR,
    C = A <3,2> COLUMNVECTOR, M = A <3,3>;
  C = C/D;
  M = (M - C * R);
END OGOS;
```



By programming M = M - C * R by submatrices, we obtain GOSM (code omitted here). Submatrix Gaussian elimination with coefficient storage can be written as [HOUS64, MCKE69]:

```
for K = 1 to N do;
   A_KK   GOS(A_KK, m);
   for J = 1 to m do;
     for i = K*m+1 to n do;
        a_i, (K-1)*m+J = a_i, (K-1)*m+J / a_(K-1)*m+J, (K-1)*m+J;
     end;
     for i = K*m+1 to n do;
       for j = (K-1)*m+J+1 to (K-1)*m+m;

         a_ij = a_ij - a_i,(K-1)*m+J * a_(K-1)*m+J,j;
       end;
     end;

     for i = (K-1)*m+J+1 to K*m do;
       for j = K*m+1 to n do;
         a_ij = a_ij - a_i,(K-1)*m+J * a_(K-1)*m+J,j;
       end;
     end;
   end;

   for I = K + 1 TO N do;
     for J1 = K + 1 TO N do;

        A_I,J1 = A_I,J1 - A_IK * A_K,J1
     end;
   end;
 end;
```

Based on this algorithm, we can write OGCSS and GOSS (code omitted here). We note that both, GOSM and GOSS can be modified by the loop reversal technique. McKeller et al [MCKE69] present a submatrix Gaussian elimination algorithm which, in our terminology, is GOSSR. In Figure 4.15, we have plotted $s(\mathcal{T})$ vs. $\mathcal{T}$ for GOS, GOSR, GOSM and GOSS. Once again, behavior similar to Cholesky factorization is observed.

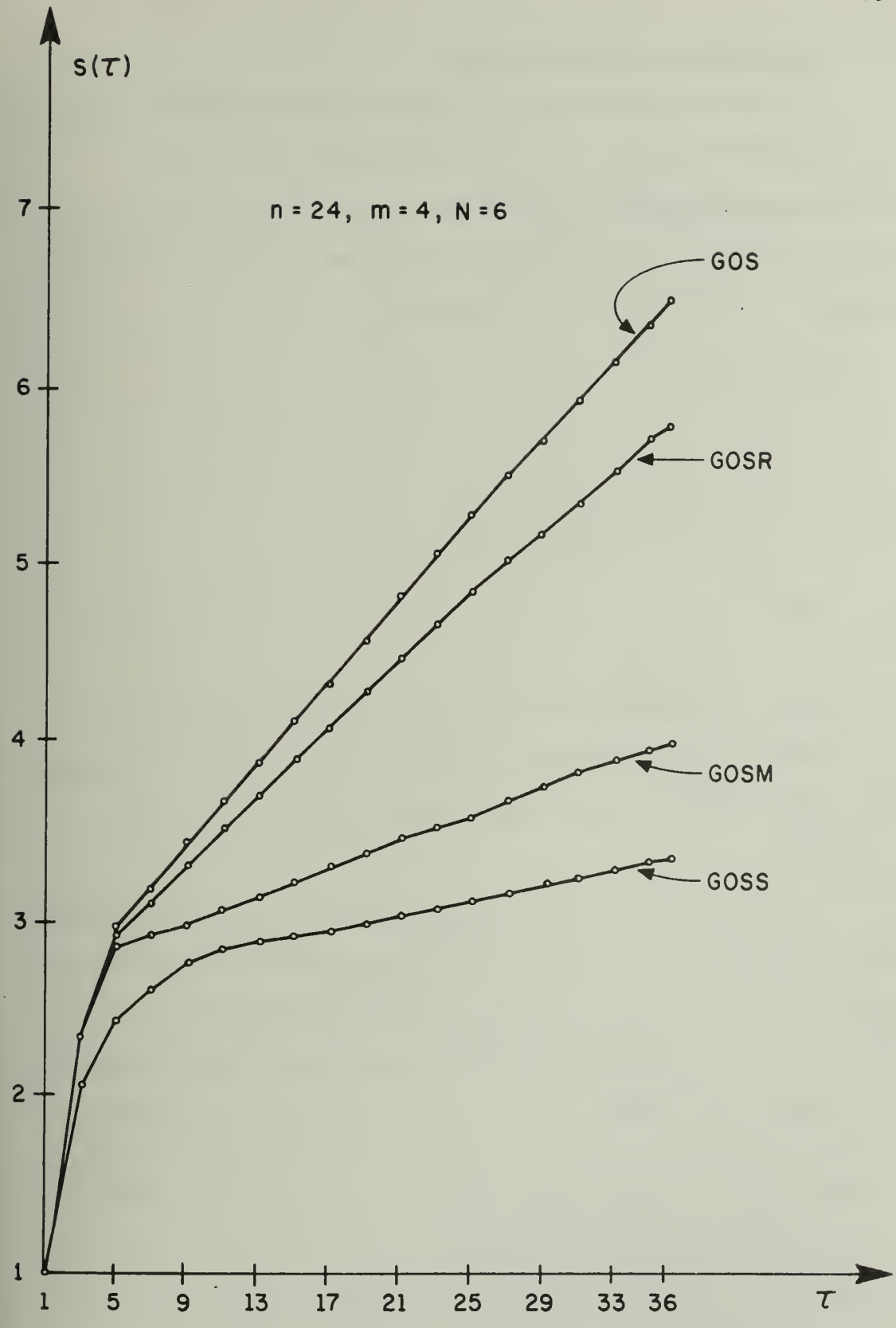


Figure 4.15. Localities of GOS, GOSR, GOSM and GOSS

4.2.4 Gram-Schmidt Orthogonalization

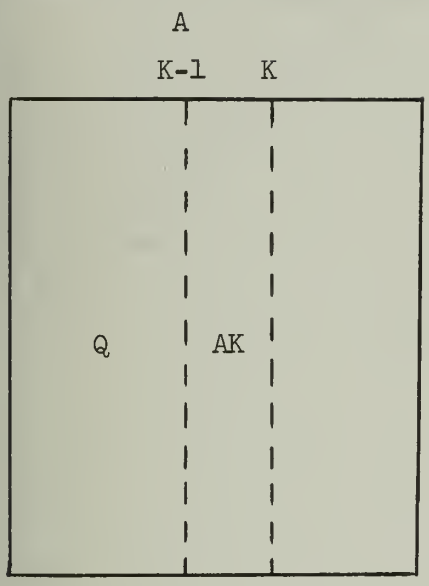
Given a generally tall $(n_1 \times n_2)$ matrix A of rank n_2 $(n_2 \leq n_1)$, its decomposition into A = B × R is known as the Gram-Schmidt orthogonalization method. Here B is an $(n_1 \times n_2)$ matrix with orthonormalized columns and R is a nonsingular upper triangular matrix of order n_2 [SCHW73]. It can be written as:

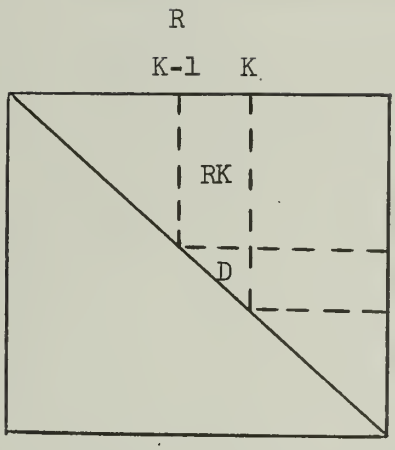
```
for k = 1 to n1 do;
  for j = 1 to k - 1 do;
                n1
     rjk = Σ   bij * aik;
           i=1
     for i = 1 to n do;
       bik = aik - rjk * bij;
     end;
  end;
                  n1
  rkk = √ Σ   b²ik    ;
          i=1
  for i = 1 to n1 do;
  bik = bik/rkk;
  end;
end;
```

A straightforward implementation of this algorithm is called ORTH (omitted here). We store B over A, thus destroying the original elements of A. A reversal technique applied to ORTH turns it into ORTHR. To identify the matrix multiplications within ORTH, we write it in OL/2 and call it OORTH.

```
OORTH:  PROC(A, R, n1, n2);
LET A BE A n1 × n2 MATRIX AND LET
R BE AN UPPER TRIANGULAR MATRIX OF ORDER (n2);
FOR K = 1, 2, ..., n2; PARTITION A, R AFTER
COLUMNS K-1, K;
SET Q = A <1>; AK = A <2>, RK = R <1,2>, D = R <2,2>;
```

```
   RK = Q' * AK;
   AK = AK - Q * RK;
   D = (AK, AK);  AK = AK/D;
END OORTH;
```

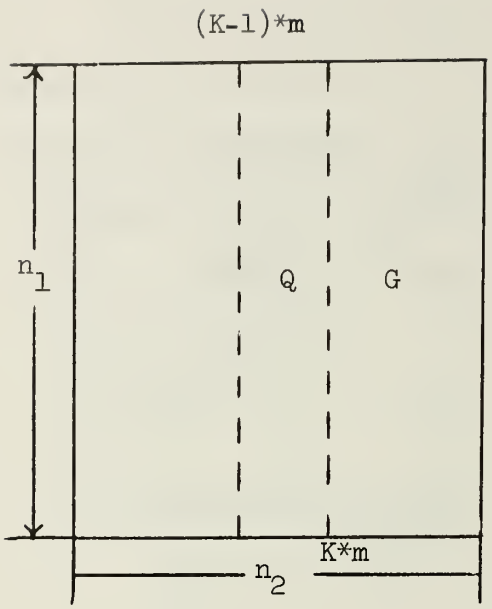


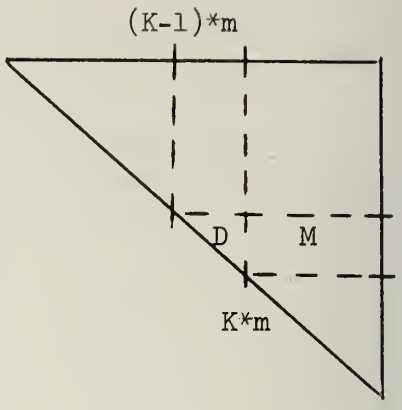


If we code the statements RK = Q' * AK and AK = AK - Q * RK by submatrices, we obtain ORTHM. Submatrix version of ORTH is [SCHO73] written in OL/2, and is called OORTHS. Note that, $N_2 = \frac{n_2}{m}$ and $N_1 = \frac{n_1}{m}$.

```
   OORTHS:  PROC,
    LET A BE n1 × n2 MATRIX AND R BE AN
    UPPER TRIANGULAR MATRIX OF ORDER (n2);
    FOR K = 1, 2, ..., N2; PARTITION A, R AFTER
    COLUMNS (K-1) * m, K * m;
   SET Q = A <2>, D = R <2,2>, M = R <2,3>, G = A <3>;

    CALL OORTH (Q, D, n1, m);
    M = Q' * G;
    G = G - Q * M;
  END OORTH;
```





This can be easily programmed in PL/I and is called ORTHS. In Figure 4.16, we have plotted $s(\mathcal{T})$ vs. $\mathcal{T}$ for ORTH, ORTHR, ORTHM and ORTHS. Once again, reversal technique can be applied to ORTHM and ORTHS but the improvement thus gained is not significant.

4.3 Conclusion

In all four algorithms considered so far, we have seen that three methods of locality improvement work very well. We have seen that a submatrix version of an algorithm improves the locality to the greatest extent. The submatrix multiplication technique for all internal multiplications is the second best technique for improvement. Reversal technique also improves locality but by a much smaller degree than the other two techniques. We also notice that, locality improving methods cannot reduce page faults for very large core allotments. The reason is that, with a large number of pages allotted, the working set of an

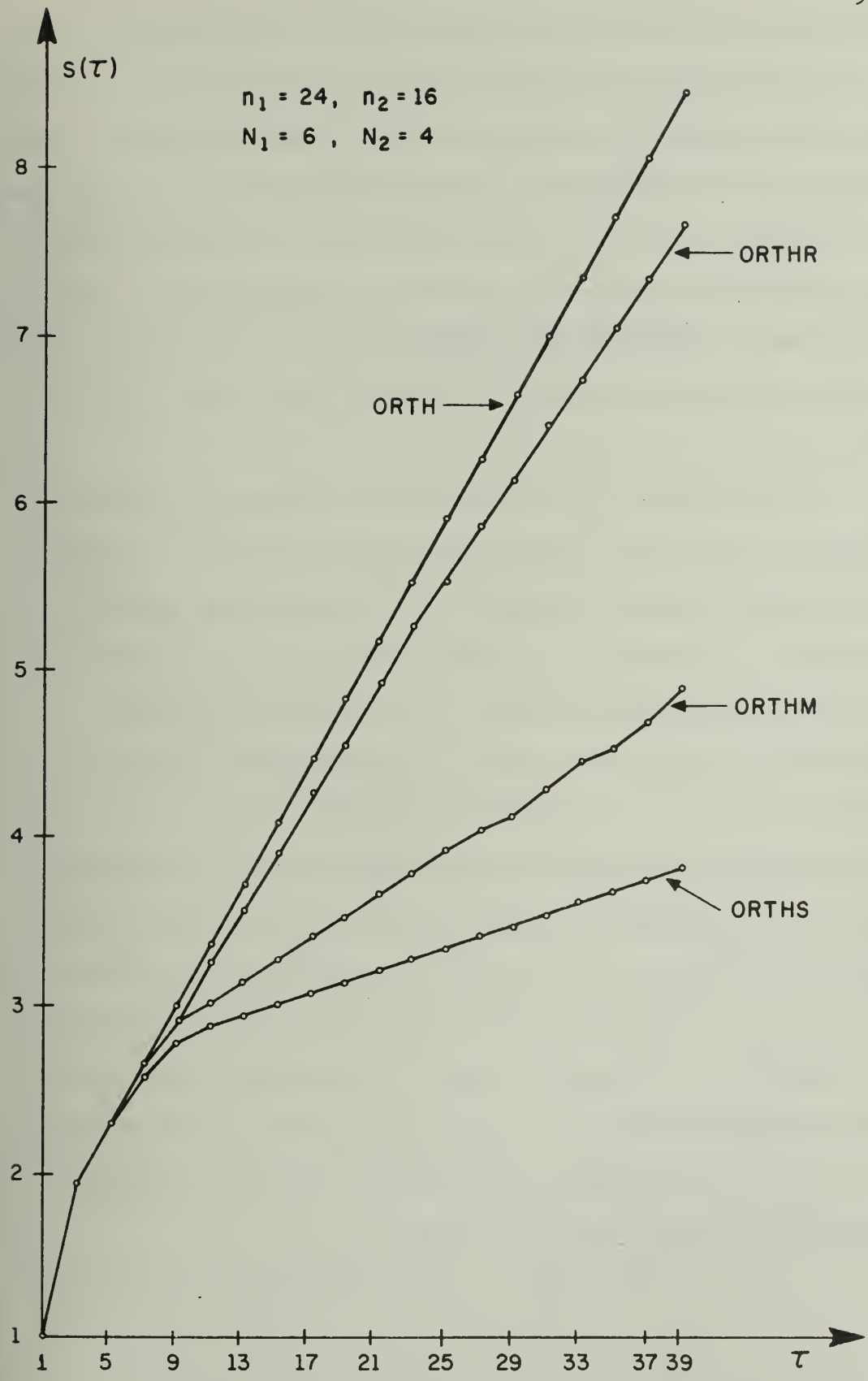


Figure 4.16. Localities of ORTH, ORTHR, ORTHM and ORTHS

ill-structured (poor locality) program can also be kept in MM. We have observed that locality improvement methods are effective when $c < \mathcal{O}(N)$. Where $\mathcal{O}(N)$ denotes kN for some constant k. Thus the asymptotic value of page faults is unaffected by locality improvements. It is also clear that the asymptotic value of page faults cannot be changed by changing paging algorithms if we restrict ourselves to demand paging. [There is no replacement required at high core allotments.] In Chapter 5, we will see that the asymptotic value of the number of page faults can be reduced by devising prepaging algorithms.

At this point, we are in a position to compare the paging performance of the three decomposition algorithms used in solution of linear systems [Cholesky decomposition, LU decomposition and Gaussian elimination]. In Figure 4.17, we have plotted $\pi(c)$ vs. c for CD, LU and GOS using LRU paging algorithm. The superiority of Cholesky decomposition is very clear. But, as is well known, it can only be used when the matrix A is symmetric and positive definite. It is also known that this algorithm has minimum operation count and has superior stability property among these three decomposition algorithms [ISAA66]. When the given matrix A is not symmetric or is not positive definite and if we have to choose between LU decomposition and Gaussian elimination, LU decomposition is preferable. Also, LU decomposition incurs lesser number of operations and is also more stable [ISAA66], from Figure 4.17, it is seen that LU decomposition is generally superior to Gaussian elimination in number of page faults also.

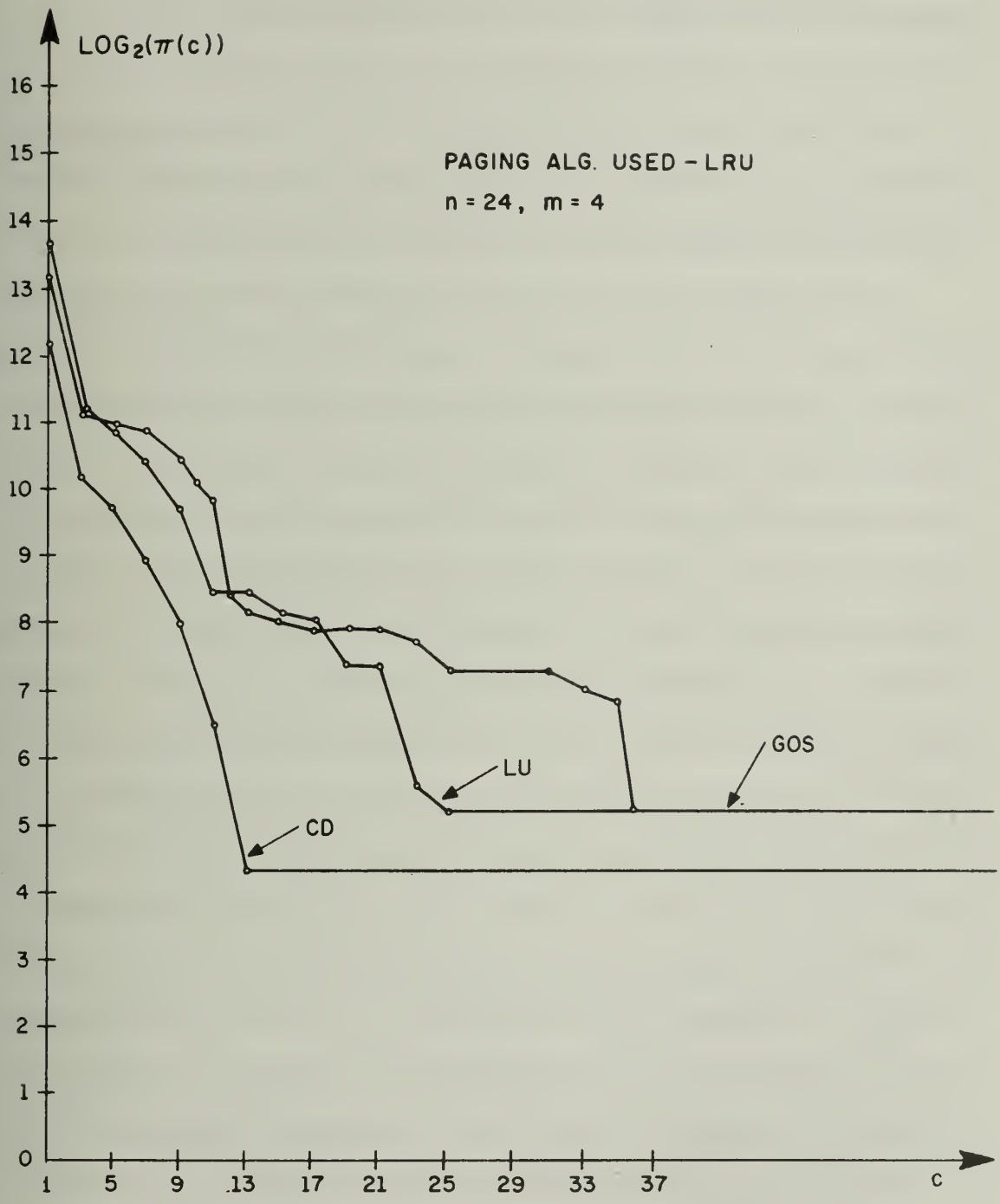


Figure 4.17. Page Faults for CD, LU and GOS

4.4 A Note About Measurement of Performance

We have combined LRU stack algorithm of Mattson et al [MATT70], WS statistics gathering algorithm of Gordon [GORD73] and multivalued MIN algorithm of Belady [BELA74] into LRU_WS_MIN algorithm. This algorithm obtains the LRU, WS and MIN statistics of a given reference string in one pass for all values of page allotments (up to the size of the address space) and for all values of the window width τ (up to a fixed maximum). We give a PL/I program for this algorithm in Appendix A.

Once a simple one pass, paging algorithm simulation is available, the next question is to easily obtain the reference strings for a given program. Since for the matrix programs that we have considered, each program has only one trace and, therefore, there is only one reference string associated with it. Also note that, we are only interested in data references and, in particular, only in array references. Given a matrix program written in PL/I, we remove all declarations. All assignment statements and expressions are reduced by removing all scalar references (and retaining only array references). Each array reference is now replaced by a call to the appropriate paging algorithm simulator (in our case LRU_WS_MIN). For each such call we must also supply the page number of the referenced page. This is determined from the name and index of the array referenced and the organization of the arrays in VA space. For example, a matrix of order n is stored from page one to page N^2 by submatrices and the submatrices are stored in a column major order, then a reference A(i, j) will be a reference to the page number $x = (N*(\lceil \frac{j}{m} \rceil - 1) + \lceil \frac{i}{m} \rceil)$, since the index of the submatrix, to which the element A(i, j) belongs, is $(\lceil \frac{i}{m} \rceil, \lceil \frac{j}{m} \rceil)$.

5. APPLICATION OF FREEING AND PREPAGING TO ARRAY PROGRAMS

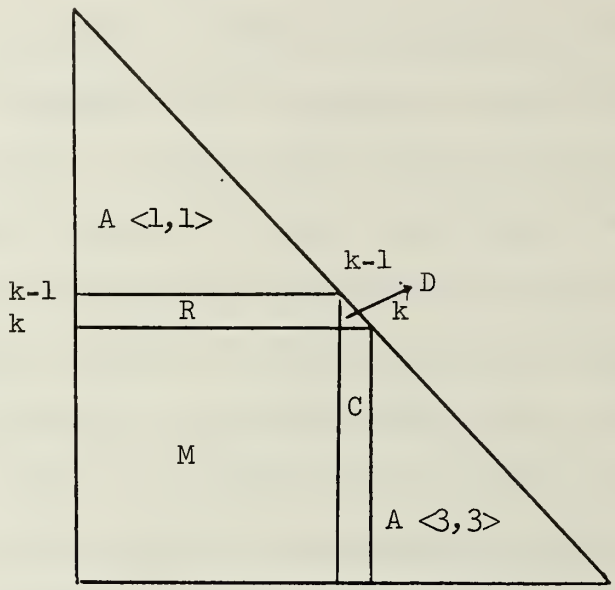
5.0 Introduction

The objective of prepaging is to reduce or avoid page faults. In many array algorithms, which are dominated by data paging, we have some inherent structure which will allow a fore-knowledge of paging needs. This can be utilized in several ways: First, if we know that a set of data pages is not required by the program in future then the space occupied by these pages in MM can be freed for some other use. Second, if we have extra space in MM, then we may be able to prefetch some of the required pages. We have noted in Chapter 4, that the locality-improvement methods reduce page faults only for small values of the page allotment c. We will see that the methods used in this chapter can help reduce page faults for larger values of c. Thus we will have covered the complete spectrum of c-values. We will show that an order of magnitude reduction in the asymptotic value of page faults can be achieved by our methods.

Once again, we start out with an example, namely, Cholesky decomposition (CD), and show how to improve its performance. We will also be concerned with measuring the improvement in the performance. Finally, in subsequent sections, we will apply the techniques to other matrix algorithms. In this chapter, we assume that the programmer is making these improvements in his program. In Chapter 6, we will discuss the question of automating these techniques.

5.1 Cholesky Decomposition

It is easier if we describe the algorithm using OL/2 and an associated diagram to identify the data-working-set. Let us consider the program OCD (refer to section 4.1) and the following partitioned array:



The algorithm OCD shows that, for a particular value of k, the subarrays R, D, M and C are referenced, but the subarrays A <1,1> and A <3,3> are not. Furthermore, we note that the elements of A <3,3> were not used in the past, but the present elements of A <3,3> will be referenced in the future. On the other hand, all the elements of A <1,1> were used in the past, therefore, they are likely to be in MM and the algorithm shows that they will not be used again. Thus A <1,1> may be marked as a dead subarray. Since the mechanism that we have, only allows pages to be declared dead, it is necessary to modify our mechanism for dead subarrays. Because of dynamic partitioning, the extent of the subarrays

vary dynamically. In this example, A <1,1> may share pages with the useful subarrays R, D, M and C. The set of shared pages is a null set whenever the condition, (k-1) ≡ 0 mod m, is satisfied. This condition corresponds to the alignment of a page boundary with the bottom boundary of the subarray A <1,1> (given by the expression k-1). Under such a condition, we can free all the pages of A <1,1>. We assume the existence of a procedure FREE(B) which frees all the pages of the subarrays, which are resident in MM. Based on this observation, OCD can be modified to yield OCDF as follows:

```
OCDF:  PROC(A, n);
         .
         .
         .
 IF MOD(k, m) = 1 THEN CALL FREE(A <1,1>);
 D = SQRT(D - (R', R'));
 C = (C - M × R')/D;
END OCDF;
```

A corresponding modification is made in the PL/I program CD (refer to section 4.1) to obtain CDF. We assume the existence of a procedure FREE(A, I, J), which declares the (I, J) submatrix of array A dead. We note that this freeing operation is carried out each time k - 1 = 0 mod m. Therefore, we need only free the bottom-most row of pages of A <1,1>. In Figure 5.1, we give the PL/I program CDF.

In Figure 5.2, we have plotted $\pi(c)$ vs. c for CD using LRU and MIN paging algorithms and for CDF using FREELRU paging algorithm.

We see that at and around point D, FREELRU behaves almost like MIN (the optimal demand paging algorithm). The page allotment for point D corresponds to the expression $\max_{k\in[1,n]} (|P(R,D,M,C)|)$ where $P(A_1, A_2, \ldots)$ denotes the pages containing the subarrays A_1, A_2, ... and

```
CDF:
 DO k = 1 TO n; K1FM = (k-1)/m;
  IF MOD(k, m) = 1 THEN
    DO J = 1 TO K1FM;
     CALL FREE(A, K1FM, J);
    END;
  S = 0;
  DO J = 1 TO k-1;
   S = S + A(k, J) * A(k, J);
  END;
  A(k, k) = SQRT(A(k, k) - S);
  DO I = 1 TO n-k;
   S = 0;
   DO J = 1 TO k-1;
    S = S + A(k+I, J) * A(k, J);
   END;
   A(k+I, k) = (A(k+I, k) - S)/A(k, k);
 END;
END CDF;
```

Figure 5.1. Cholesky Decomposition with Freeing

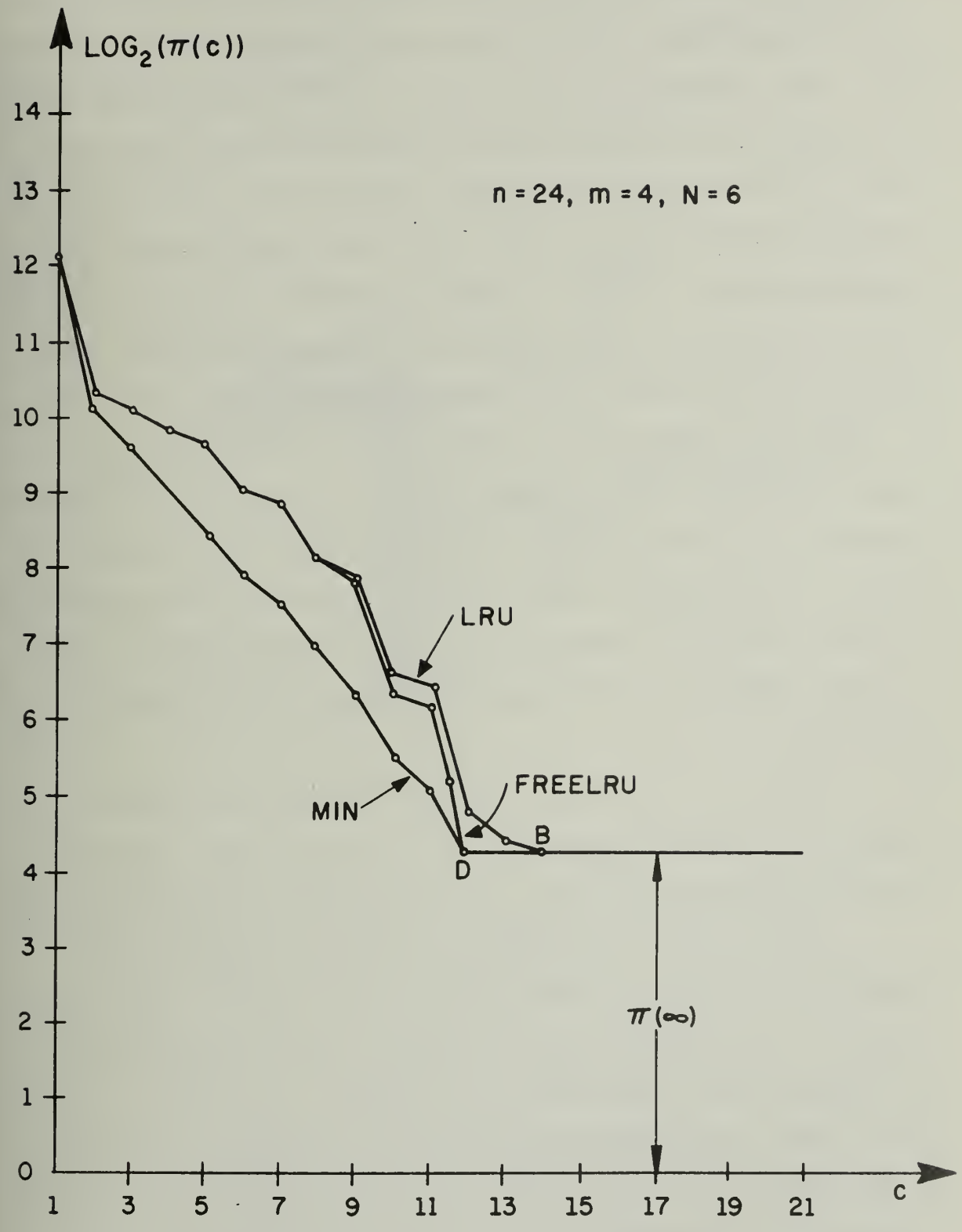


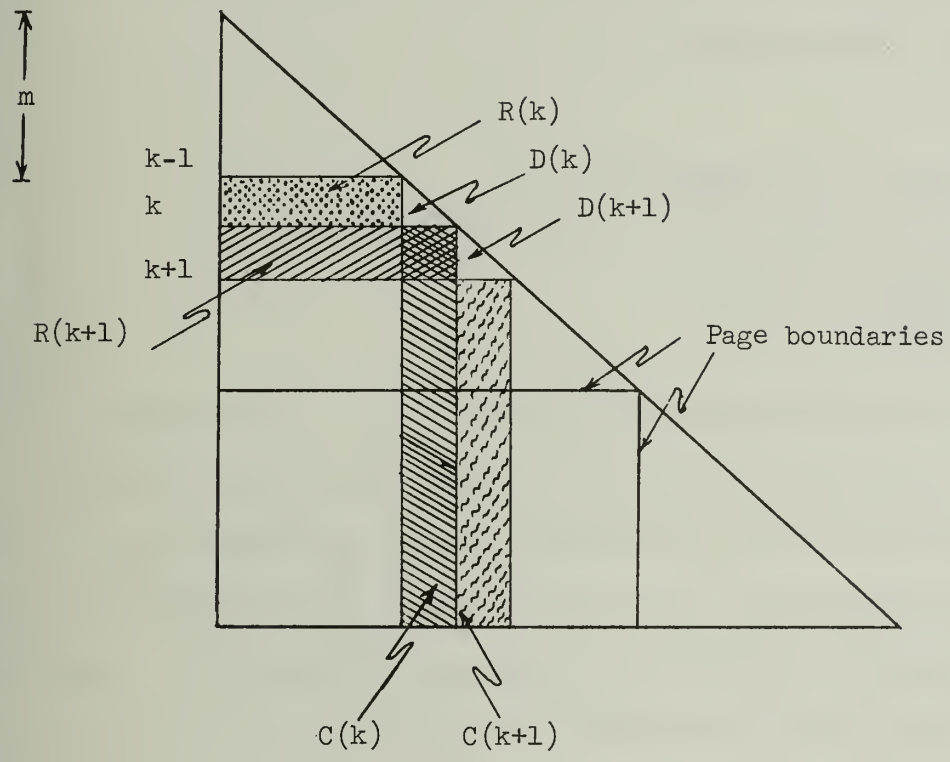
Figure 5.2. Page Faults Using LRU, FREELRU and MIN Algorithms

|P| denotes the number of pages in the set P. Clearly, this is the same as the maximum number of active pages during the execution of the program CD. This is easily found to be equal to $\left\lfloor \frac{(N+1)^2}{4} \right\rfloor$ which in the case N = 6 reduces to 12. We note that the asymptotic value of page faults, i.e., $\pi(\infty)$, is not changed by freeing. Note that $\pi(\infty)$ = total number of distinct pages referenced = size of the address space (in case each page is referenced) = $\frac{N(N+1)}{2}$ (= 21 in the present case). We note that for very low values of c, freeing does not reduce page faults because dead pages are removed anyway by the replacement action of LRU. Only after c = $\geq 6(N)$ then freeing starts to help. For very high values of c (i.e., $6(N^2)$), once again, freeing does not help since the whole of the VA space can be stored in MM and no page replacements are necessary. Because of simulation time limits we have not carried out our experiments with large values of the matrix size. It is hoped that with large value of n (the matrix size) the advantages of freeing will be brought out more clearly.

We note that the freeing technique can be applied to other variations of Cholesky decomposition (e.g., CDR, CDM, CDS). Since the nature of improvements is similar, we do not discuss these further.

Next we proceed to investigate what can be prepaged in OCD. We know that the four useful subarrays are: R, C, M and D. Since the extent of these subarrays change with the loop control variable k, we will indicate this by subscripting these subarrays. Note that, $R(k+1) \subseteq M(k) \cup C(k)$ and $M(k+1) \subseteq M(k) \cup C(k)$. Therefore, if we decide to look only one loop execution ahead, we need to prepage the subarrays C and D only. We note that, $P(C(k+1) = P(C(k))$ <u>if</u> $k \not\equiv 0 \bmod m$. Here

P(B) denotes the pages containing the subarray B. Similarly, P(D(k+1)) = P(D(k)) if k ≠ 0 mod m. The reason for this can be seen from the following diagram:



Therefore, we need to prepage only when the condition k = 0 mod m is satisfied. Under such a condition we prepage C(k+1) and D(k+1).

If we decide to prepage the newly required elements of C and D for each new execution of the loop just before beginning the execution of the loop, then we will prepage C(k) and D(k) if k ≡ 1 mod m. Since we are going to consider only demand prepaging algorithms, it does not matter when we prepage a certain set of pages, so long as the prepaging is done prior to a reference to any one of these pages. Assume that we

have issued a prepage statement for a set of pages P at time t. Assume further that at $t' > t$, $r_{t'} = i \in P$, then and at t' we have a page fault. At this time page i will be fetched and as many pages of P will be prefetched as will be allowed by the number of empty pages in MM. With this modification, we obtain the following algorithm:

```
OCDFP:  PROC(A, n, m);
          .
          .
          .
          .
 IF MOD(k, m) = 1 THEN
   DO; CALL FREE (A <1,1>); CALL PREPAGE(C, D);
   END;
 D = SQRT(D - (R', R'));
 C = (C - M × R')/D;
END OCDFP;
```

We can also modify CDF to obtain CDFP in a similar way. In Figure 5.3, we have plotted $\pi(c)$ vs. c curve for CD using LRU paging algorithm, for CDFP using FREEDPRE4LRU paging algorithm and for CD using DPMIN paging algorithm. We see that prepaging using FREEDPRE4LRU (in fact, demand prepaging) reduces the page faulting considerably for large values of c. In particular, the asymptotic value of π has been brought down from $\frac{N(N+1)}{2}$ to N. This is one order of magnitude reduction. We see that the asymptotic value of π for DPMIN is 1. We note here that such a prepaging technique can also be applied to other variations of Cholesky factorization (namely, CDR, CDM and CDS) but we omit this here, since the results are very similar to that of CD. Since with prepaging, the number of page pulls is different from the number of page faults, we need to consider these two measures separately. In Figure 5.4, we have plotted, the number of page pulls vs. c for CD using LRU, MIN and DPMIN paging algorithms and for CDFP using FREEDPRE4LRU paging algorithm. We see that, the number page pulls for FREEDPRE4LRU is generally lower than the page pulls for LRU and only

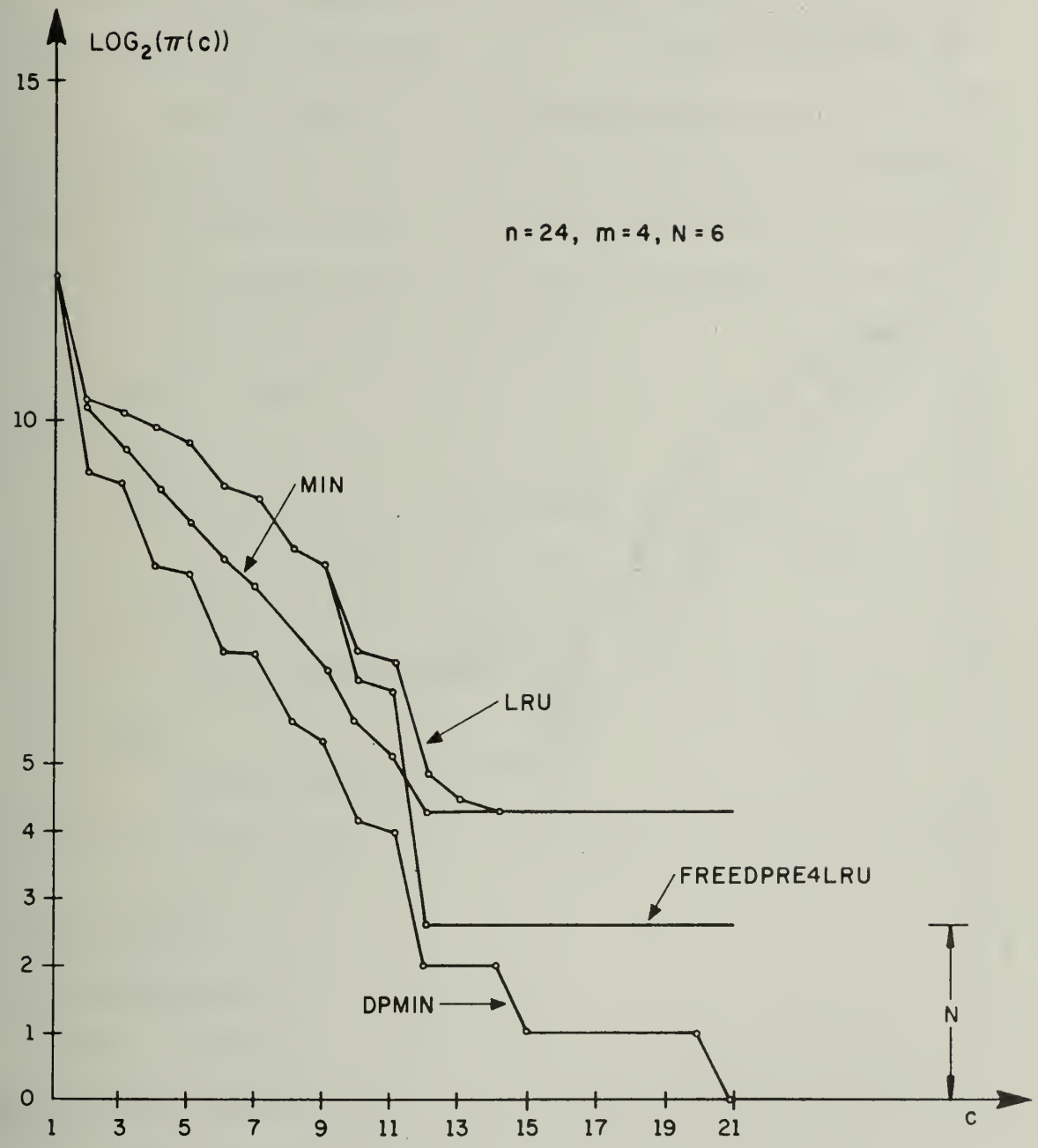


Figure 5.3. Page Faults Using LRU, FREEDPRE4LRU, MIN and DPMIN Paging Algorithms

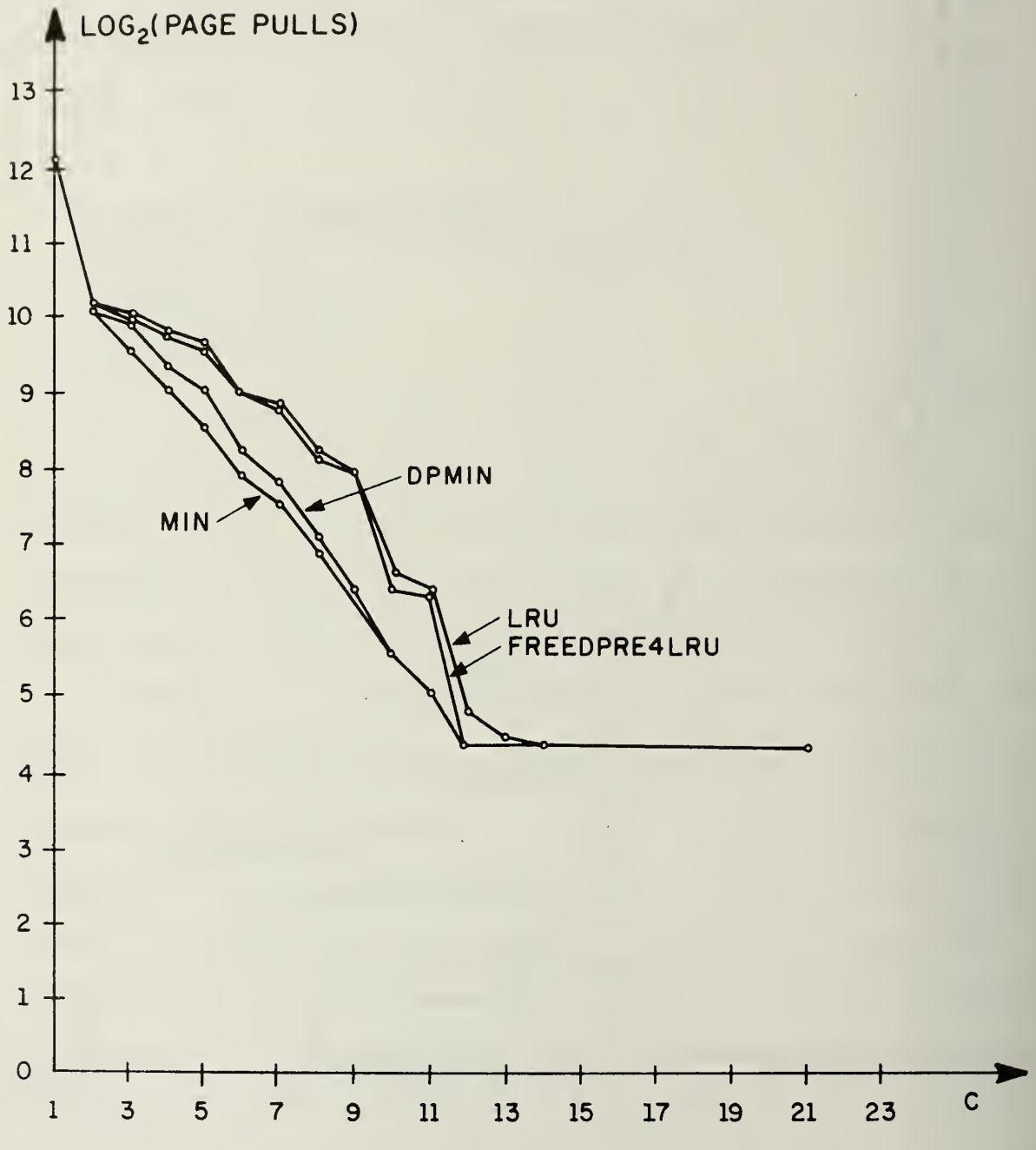


Figure 5.4. Page Pulls Using LRU, FREEDPRE4LRU, MIN and DPMIN Paging Algorithms

rarely does it go above and then only by a very small amount. We see that the page pulls for DPMIN and MIN are generally equal except in some cases where DPMIN incurs more page pulls than MIN. Note that the asymptotic value of page pulls is the same for all cases and is equal to the size of the data address space (= N(N+1)/2).

We can view the process of freeing and prepaging done in CD in a slightly different way. Let us denote the data-working set of one loop execution of CD by DWS(k), where k is the loop control index. In words, DWS(k) is the set of pages referenced in the k^{th} execution of the loop. DWS(k) can be recursively defined as follows:

a) DWS(1) = P(C(1)) ∪ P(D(1))

b) If k ≡ 1 mod m then

DWS(k) = DWS(k-1) ∪ P(D(k))

∪ P(C(k)) - P(A<1,1>(k) - A<1,1>(k-1))

= DWS(k-1) ∪ P(D(k)) ∪ P(C(k))

- P(R(k-1)) - P(D(k-1))

else

DWS(k) = DWS(k-1);

This knowledge of DWS allows us to do the appropriate freeing and prepaging.

5.2 Other Examples

5.2.1 LU Decomposition

For LU decomposition (refer to OLU and the associated diagram in section 4.2), we have

a) DWS(1) = P(Y(1)) ∪ P(Z(1));

b) IF $k \equiv 1 \bmod m$ then

$DWS(k) = DWS(k-1) \cup P(Y(k)) \cup P(Z(k))$

$- P(A\langle 1,1\rangle(k-1))$

$= DWS(k-1) \cup P(Y(k)) \cup P(Z(k)$

$- P(R(k-1)) - P(X(k-1)) - P(D(k-1));$

else

$DWS(k) = DWS(k-1);$

Based on this observation, LU and OLU can be modified to free $A\langle 1,1\rangle$ and prepage Y and Z. We do not present the detailed programs LUF, LUFP, OLUF and OLUFP. In Figure 5.5, we have plotted $\pi(c)$ vs. c for LU using LRU, MIN and DPMIN paging algorithms, for LUF using FREEDPRE4LRU paging algorithm. We see that freeing and prepaging helps only for larger values of c and that the asymptotic value of $\pi(c)$ has been brought down from N^2 to N.

5.2.2 Gaussian Elimination

For Gaussian elimination (referring to OGOS and its associated diagram in section 4.2), we have,

a) DWS(1) = P(A); i.e., the whole array A.

b) If $k \equiv 1 \bmod m$ then

$DWS(k) = DWS(k-1)$

$- P(A\langle 1,1\rangle \cup A\langle 1,2\rangle \cup A\langle 1,3\rangle \cup A\langle 2,1\rangle$

$\cup A\langle 3,1\rangle).$

$= DWS(k-1) - P(C(k-1)) - P(D(k-1)) - P(R(k-1));$

else

$DWS(k) = DWS(k-1);$

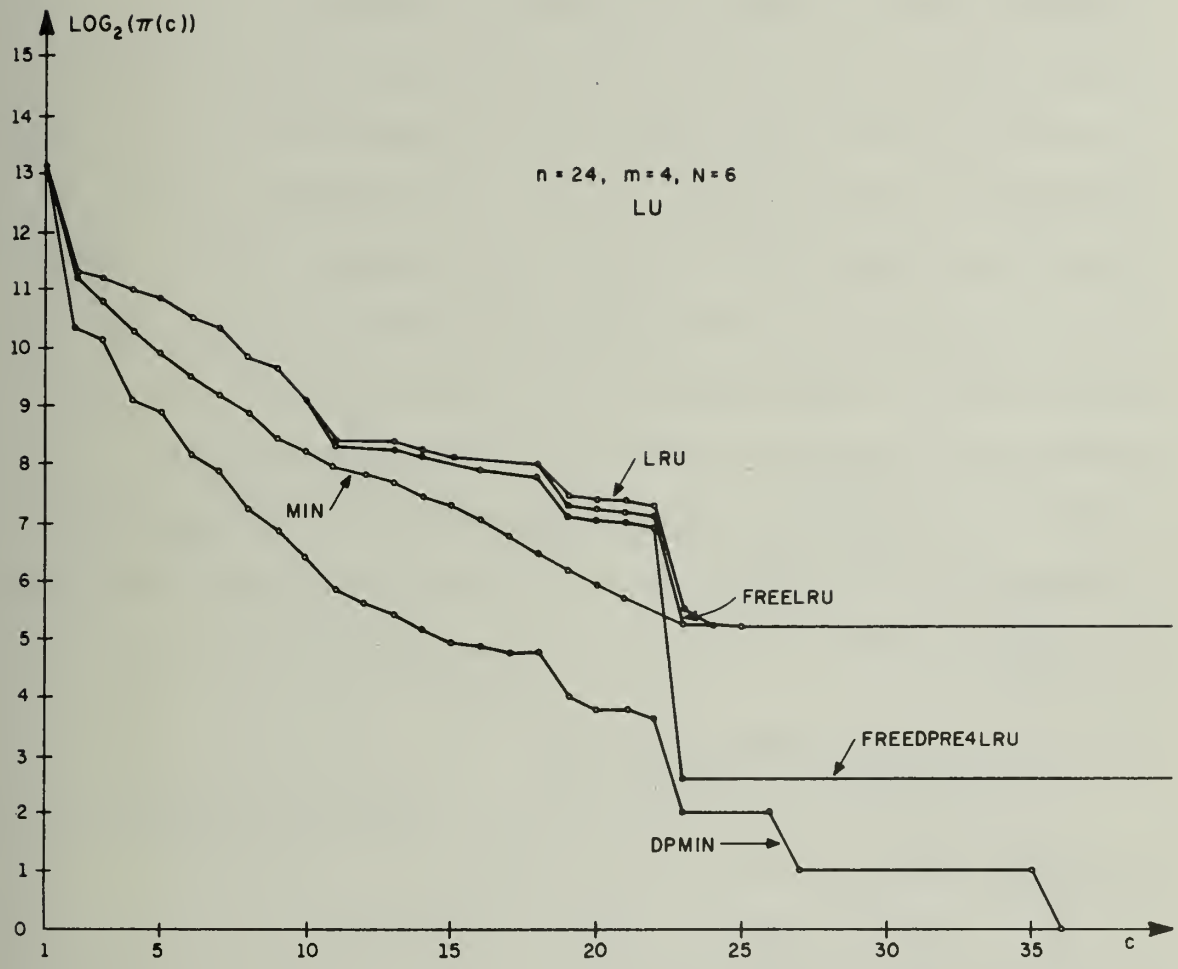


Figure 5.5. Effect of Prepaging on LU Decomposition

From these equations, we see that only prepaging that can be done is outside (before) the loop. And since whole of the array A is to be prepaged only a limited number of pages as allowed by the available space will really be prefetched. Freeing of the subarray A<1,1>, A<1,2>, A<1,3>, A<2,3> and A<3,1> can be done within the loop when the condition, $k \equiv 1 \bmod m$, is met. With these modifications we have GOSF, GOSFP, OGOSF, and OGOSFP (we omit the corresponding programs). In Figure 5.6, we have plotted $\pi(c)$ vs. c for GOS using LRU, MIN and DPMIN paging algorithms, for GOSF using FREELRU paging algorithm and for GOSFP using FREEDPRE4LRU paging algorithm. We note that $\pi(\text{FREELRU}) = \pi(\text{LRU})$. In other words, freeing does not help in this algorithm. Reason for this is that dead page is precisely the least recently used page. Thus the page replaced by FREELRU and LRU is the same at any time. The same effect is not necessarily observed with other paging algorithms. For example if we were using FIFO instead of LRU, then in all probability, $\pi(\text{FREEFIFO}) < \pi(\text{FIFO})$ for GOS. We also note that, prepaging and freeing does help and reduces the asymptotic value of $\pi(c)$ from N^2 to 1.

5.2.3 Gram-Schmidt Orthogonalization

For Gram-Schmidt orthogonalization (referring to OORTH and its associated diagram in section 4.2), we have,

a) DWS(1) = P(D(1)) ∪ P(RK(1)) ∪ P(AK(1));

b) If $k \equiv 1 \bmod m$ then

DWS(k) = DWS(k-1) ∪ P(AK(k)) ∪ P(RK(k))

∪ P(D(k)) - P(R<1,1>(k))

= DWS(k-1) ∪ P(AK(k)) ∪ P(RK(k) ∪ D(k))

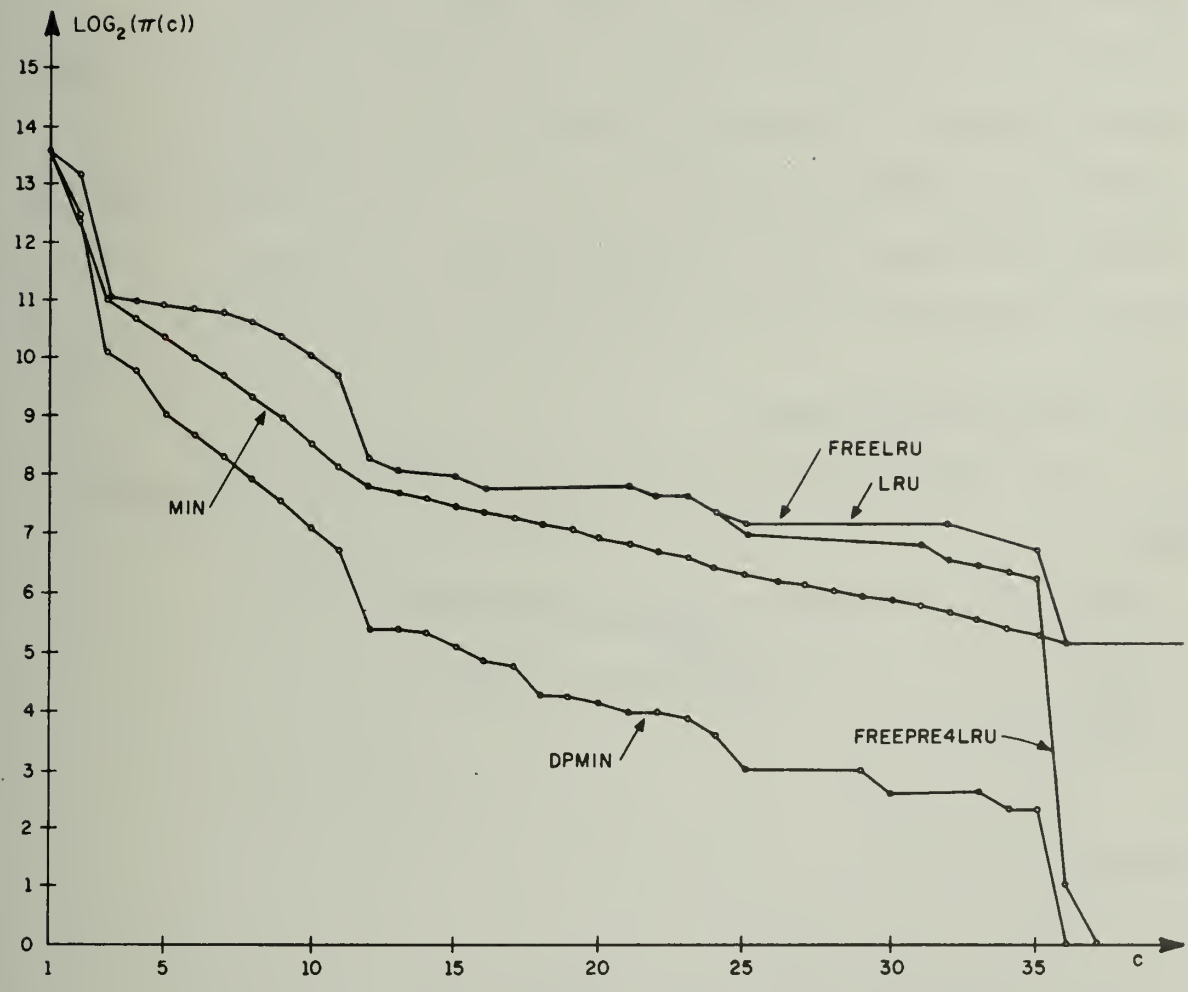


Figure 5.6. Effect of Prepaging on Gaussian Elimination

- P(RK(k-1) ∪ D(k-1));

else

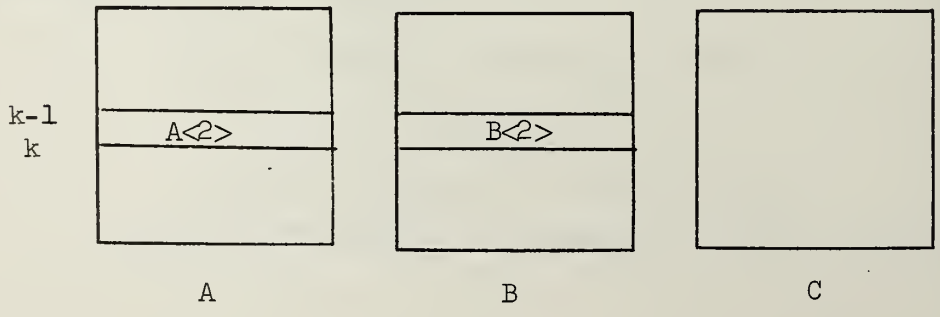
DWS(k) = DWS(k-1);

From these equations, we see that, when the condition k ≡ 1 mod m (within the loop just after the backward branch target), we can free the subarray R<1,1> and prepage the subarrays AK, RK and D. With appropriate modifications, we obtain ORTHF, ORTHFP, OORTHF, and OORTHFP. In Figure 5.7, we have plotted π(c) vs. c for ORTH using LRU, MIN and DPMIN paging algorithms, for ORTHF using FREELRU paging algorithm and for ORTHFP using FREEDPRE4LRU paging algorithm. We see, once again, that freeing and prepaging helps for larger values of c and that it brings down the asymptotic value of π(c) from $(N_1N_2 + N_2(N_2+1)/2)$ to N_2.

5.2.4 Matrix Multiplication

For matrix multiplication, consider the following program OMM:

```
OMM:
   LET A, B, C BE MATRICES OF ORDER (n);
   FOR k = 1, 2, ..., n-1; PARTITION A, B AFTER
   ROWS k-1, k;
      A<2> = B<2> * C;
END OMM;
```



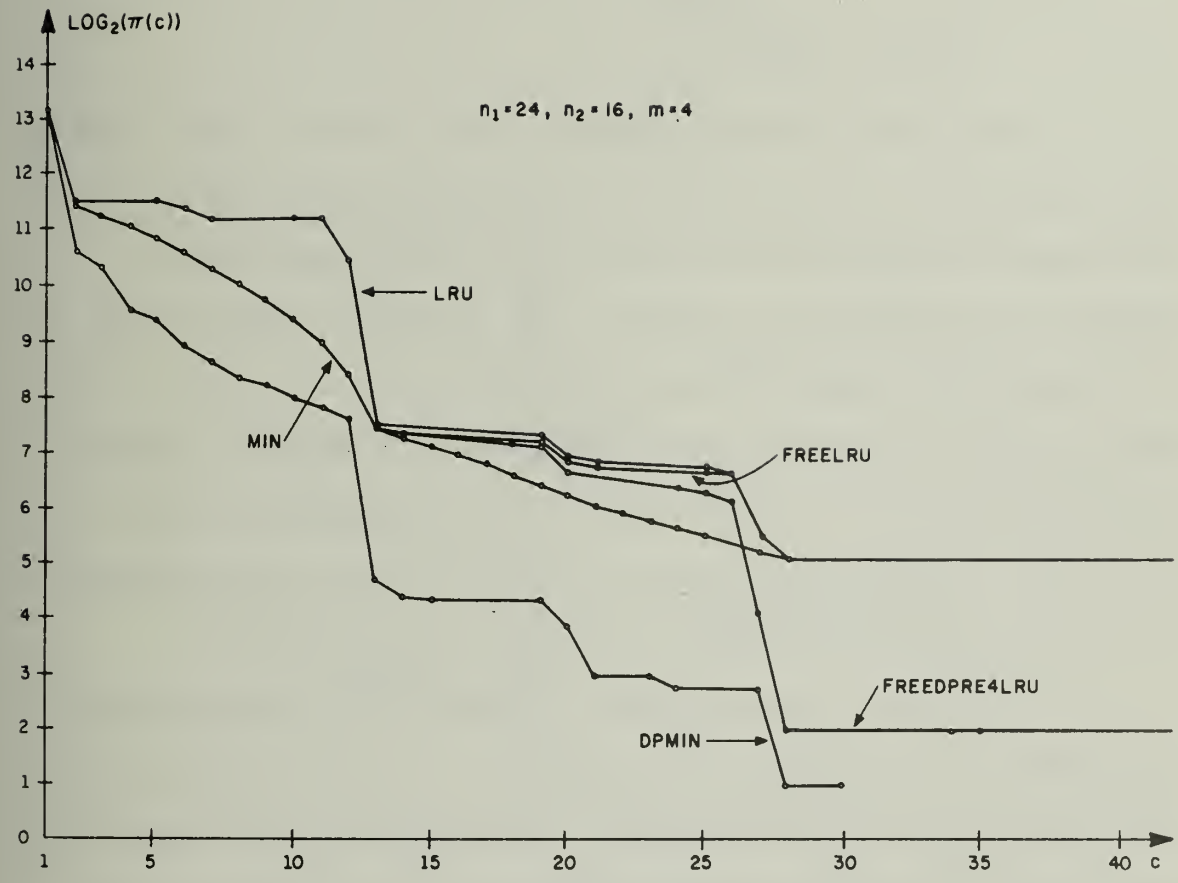


Figure 5.7. Effect of Prepaging on Orthonormalization

[Note that the above can easily be programmed in OL/2 as the primitive A = B * C but we have written OMM for obtaining DWS.] From the above, we have,

a) DWS(1) = P(C) ∪ P(B<2>(1)) ∪ P(A<2>(1));

b) If k ≡ 1 mod m then

DWS(k) = DWS(k-1) - P(B<1>(k))

- P(A<1>(k)) ∪ P(B<2>(k)) ∪ P(A<2>(k))

= DWS(k-1) - P(B<2>(k-1)) - P(A<2>(k-1))

P(B<2>(k)) ∪ P(A<2>(k));

else

DWS(k) = DWS(k-1);

Based on this, we must prepage the whole array C outside (before) the loop body and inside the loop, we can free subarrays B<1> and A<1> and prepage subarrays B<2> and A<2> when the loop control variable satisfies the condition k ≡ 1 mod m. With appropriate modifications, we obtain MMF and MMFP. In Figure 5.8, we have plotted $\pi(c)$ vs. c for MM using LRU, MIN and DPMIN paging algorithms, for MMF using FREELRU paging algorithm and for MMFP using FREEDPRE4LRU paging algorithm. Once again, we see that for large values of c, freeing and prepaging helps and that it reduces the asymptotic value of $\pi(c)$ from $3N^2$ to N.

We have seen that our methods of freeing and prepaging improve performance for large values of c ($> \Theta(N)$) and that the methods work for many different matrix algorithms. The method owes its success to our ability to identify the data-working-sets of the programs. This identification is possible, because of the well structured data-referencing

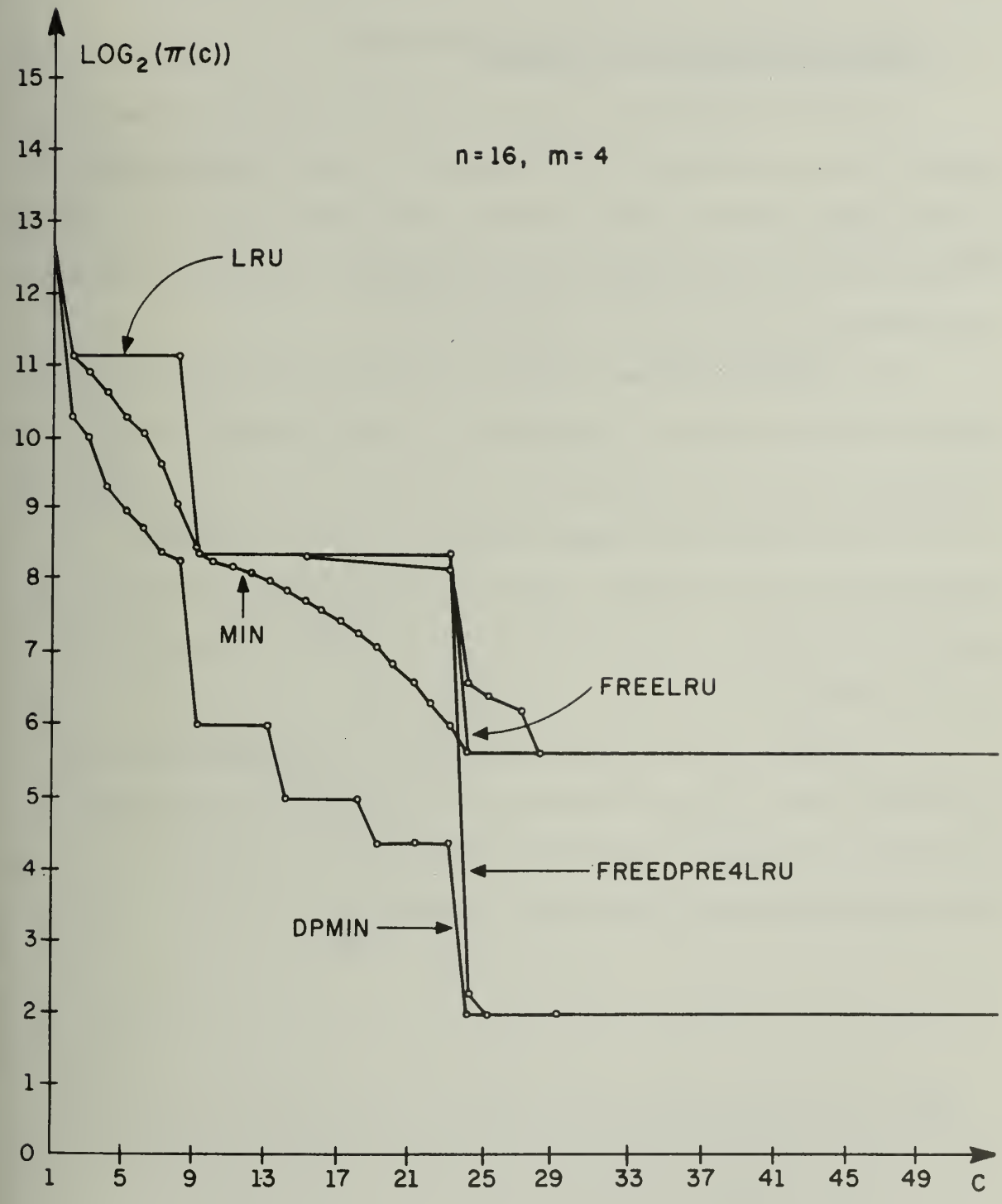


Figure 5.8. Effect of Prepaging on Matrix Multiplication

of matrix algorithms. In Chapter 6, we will investigate whether the identification of data-working-sets can be done by an optimizing compiler.

5.3 Combined Effect of Locality Improvement and Working-Set Identification Methods

We have noted that the locality improvement methods of Chapter 4 improve paging performance for low values of c ($< \mathcal{O}(N)$) and the methods of this chapter improve paging performance for higher values of c ($> \mathcal{O}(N)$). In this section we will combine the two methods and measure the resultant improvements.

The version of Cholesky factorization without any improvements is CD and the version with all improvements is CDSRFP. We give a PL/I program for the latter in Appendix C. In Figure 5.9, we have plotted $\pi(c)$ vs. c for CD using LRU paging algorithm and for CDSRFP using FREEDPRE4LRU paging algorithm. The vast improvement in performance needs no elaboration. We have also plotted the numberof page pulls vs c for CDSRFP.

In Figure 5.10, 5.11, 5.12 and 5.13 we draw similar curves for LU decomposition, Gaussian elimination, Gram-Schmidt orthogonalization and matrix multiplication respectively. Fully improved versions of these algorithms are also presented in Appendix C.

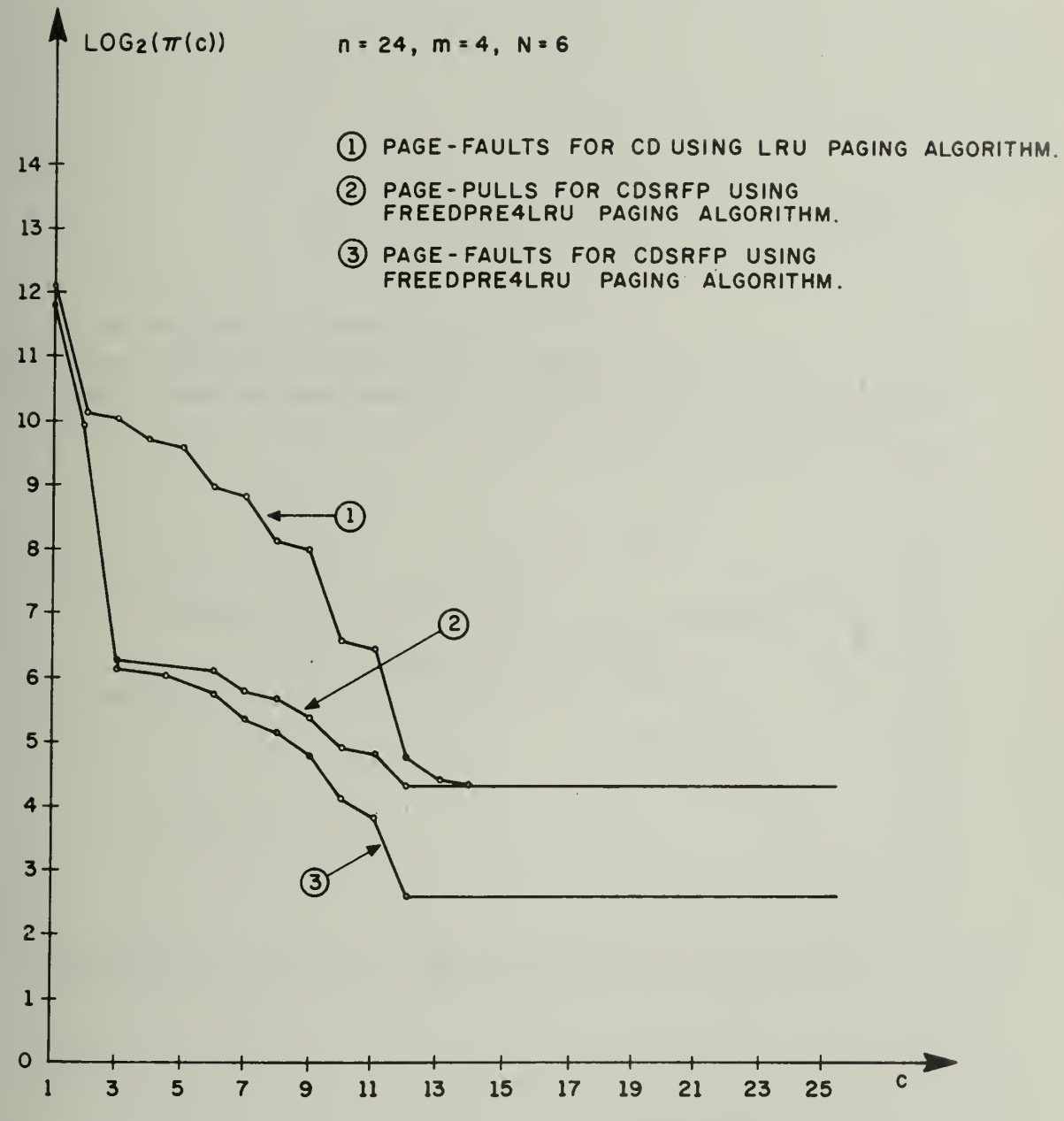


Figure 5.9. Page Faults for CD and CDSRFP

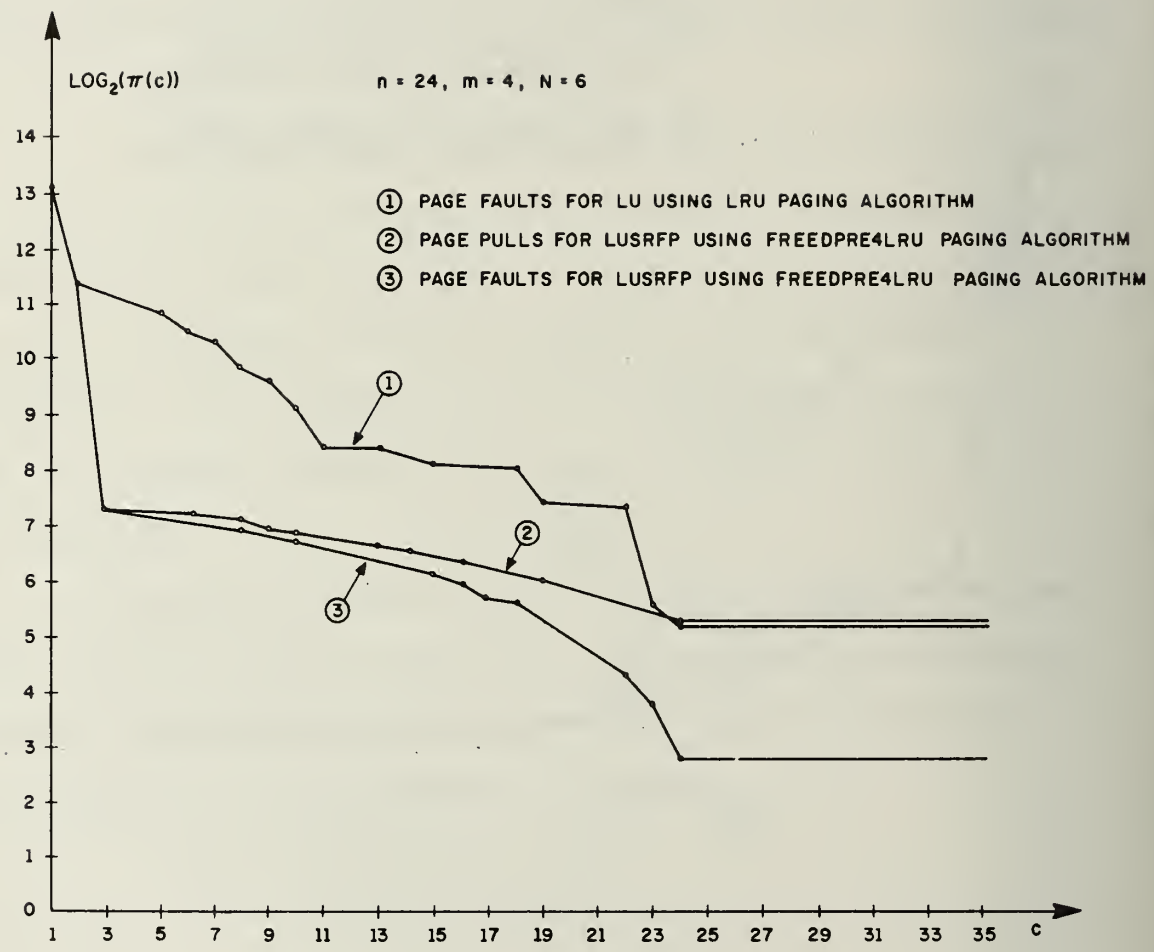


Figure 5.10. Page Faults for LU and LUSRFP

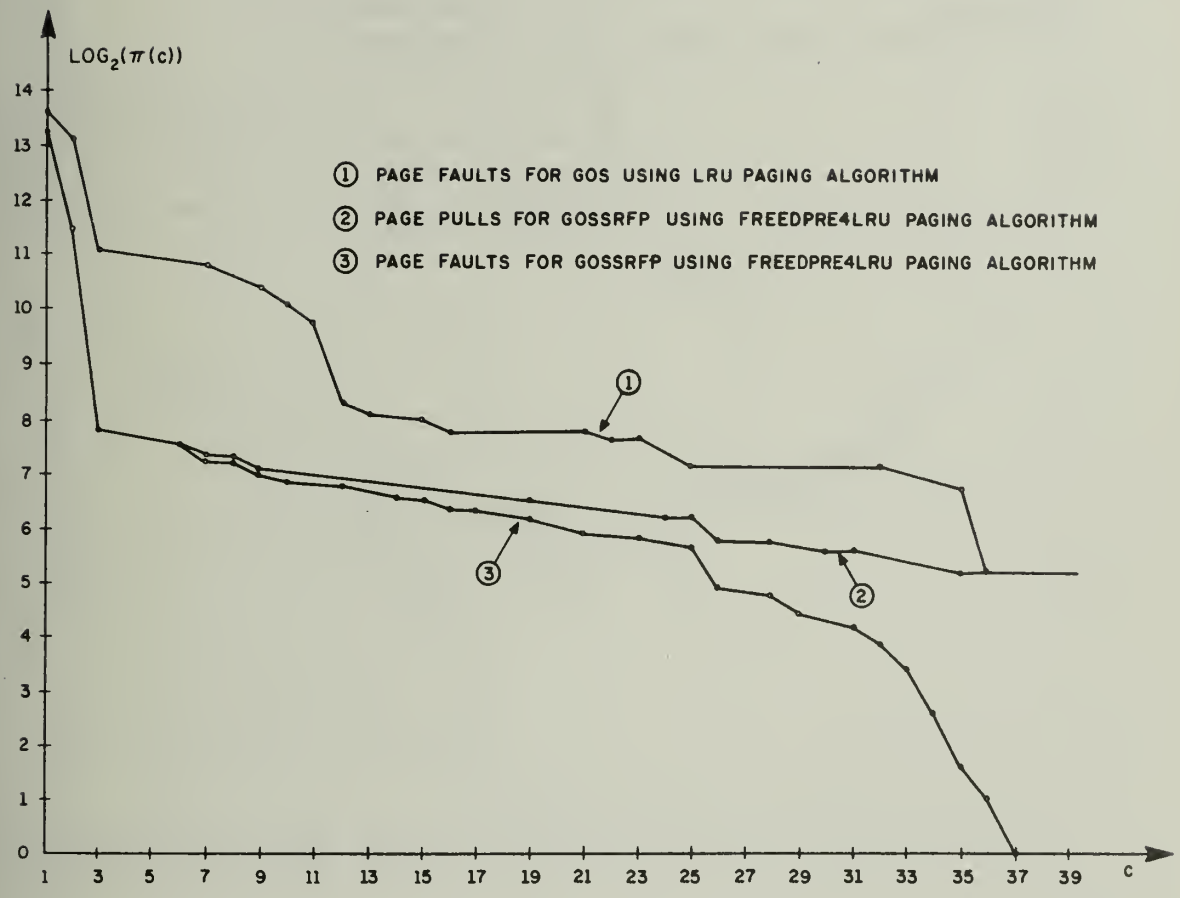


Figure 5.11. Page Faults for GOS and GOSSRFP

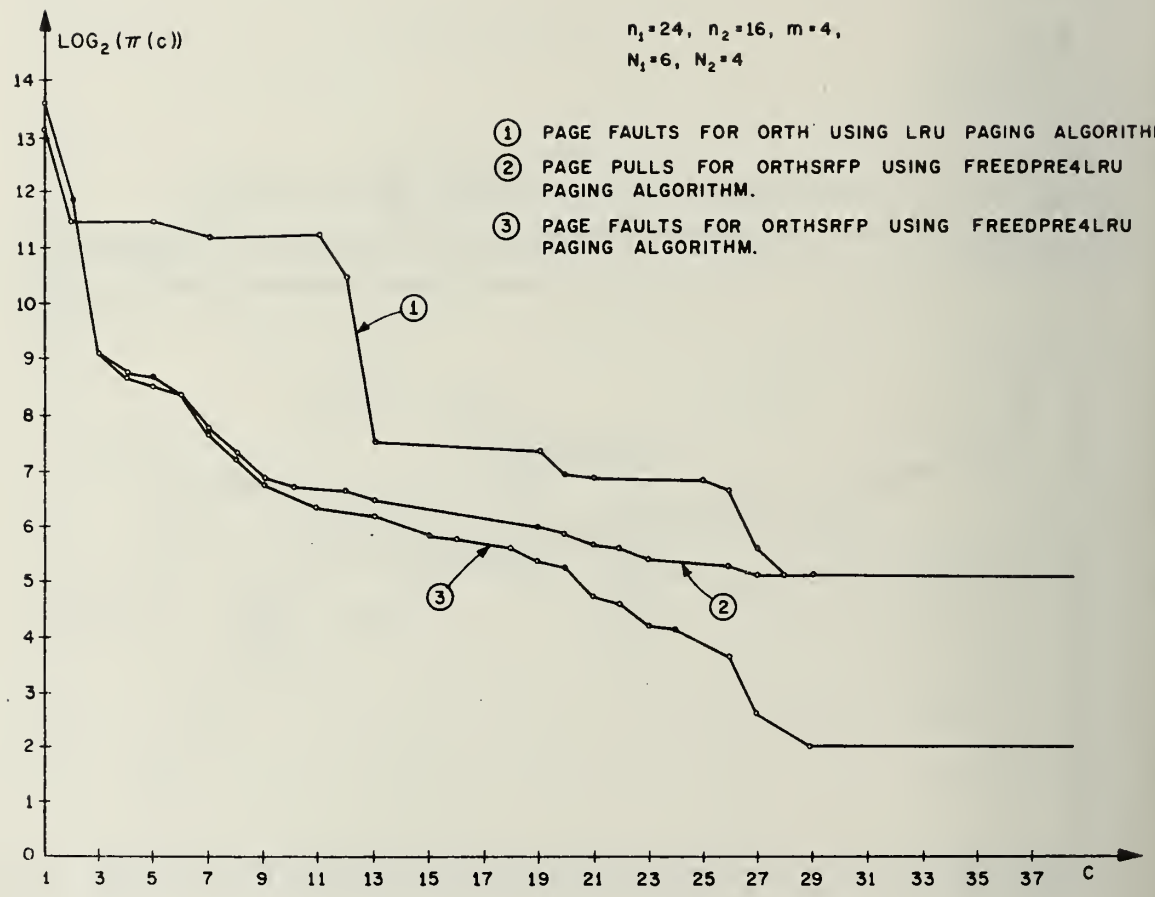


Figure 5.12. Page Faults for ORTH and ORTHSRFP

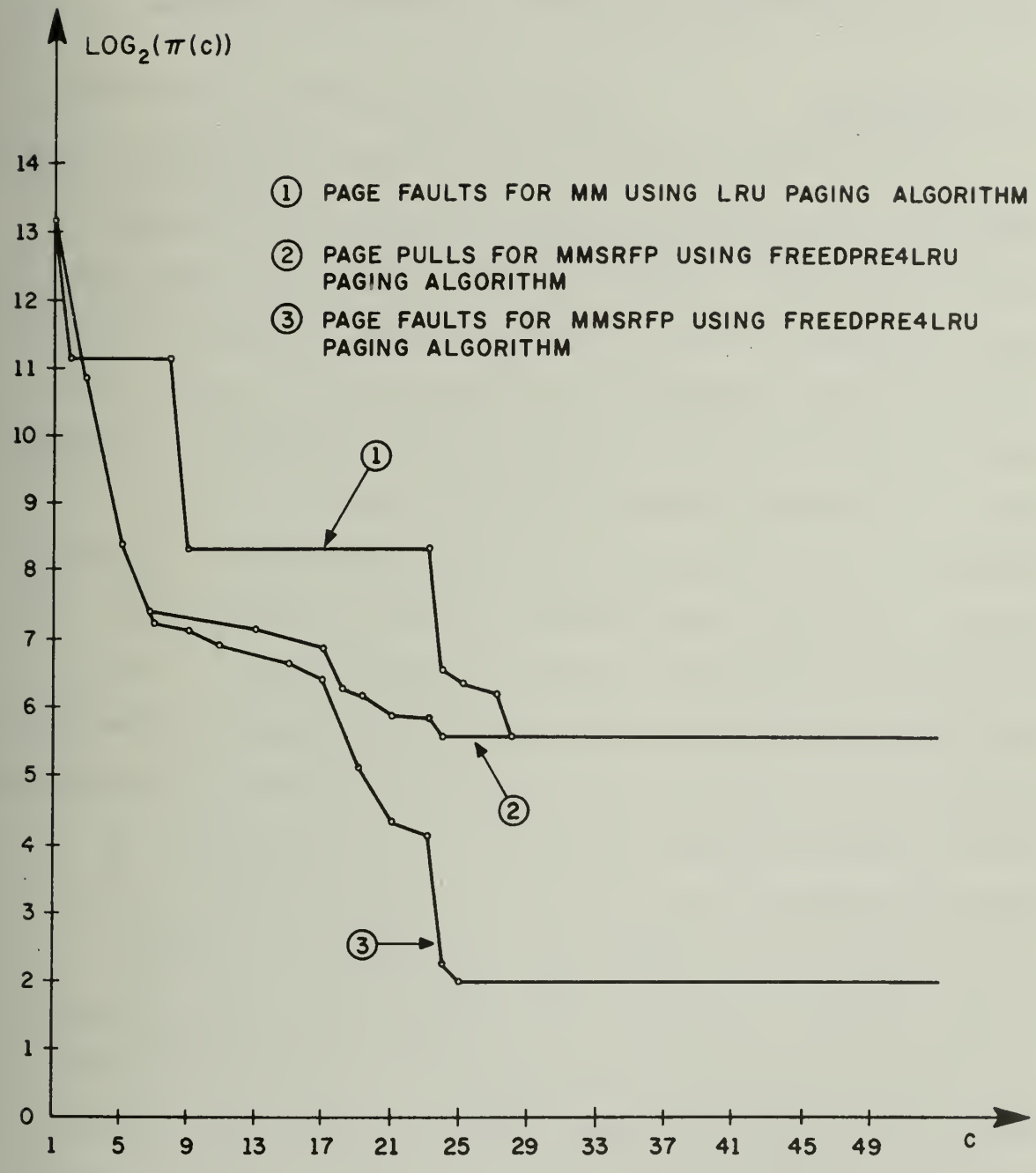


Figure 5.13. Page Faults for MM and MMSRFP

6. AUTOMATION OF PERFORMANCE IMPROVEMENT TECHNIQUES

6.0 Introduction

We have shown, in Chapter 4, how to improve the locality of matrix algorithms, and in Chapter 5, we have shown how to include freeing and prepaging techniques. We have seen that these techniques improve the paging performance of matrix algorithms. However, in the previous chapters, we have implicitly assumed that the programmer has the responsibility to modify his program to obtain the improved paging performance. In general, they are not capable or willing to make these changes; neither is it desirable to detract the user from his main purpose--namely, to solve his problem. Therefore, we will consider how one is to automate these techniques by incorporating them into the computer system.

The methods that we have presented clearly require an advanced knowledge of the program's reference pattern, which means that the operating system contains insufficient knowledge of the reference string. An optimizing compiler, on the other hand, may be able to gather enough information about the reference pattern and may be able to make the necessary changes to obtain the improved paging performance. In section 6.1, we will sketch several transformations of program segments which improve the locality of the segment, and we will give simple conditions for these transformations to be valid. In section 6.2, we will present an algorithm which will abstract information from an OL/2 program and then, later, use this information to do freeing and prepaging for the

program. It is clear that a similar algorithm can be given for an element (PL/I like) source language, but the amount of information to be processed will be very large so as to be prohibitive in the amount of overhead incurred.

6.1 Transformations Which Improve Locality

6.1.1 Loop Reversal

First we start out with an element language like PL/I. Consider a program segment P as follows:

```
P:  DO I = 1 TO n1;
      .
      .
    DO J = 1 TO n2;
     S1
     S2
      .
      .
    END;
      .
      .
   END P;
```

Now consider a transformation T, of the program segment P into the program segment P'.

```
P': DO I = 1 TO n1;
      .
      .
    IF MOD(I,2) = 1 THEN
     DO; JLL=1; JHL=n2; JSTEP=1 ; END; ELSE ;
     DO; JLL=n2; JHL=1; JSTEP=-1;END;
   DO J=JLL TO JHL BY JSTEP;
    S1
    S2
     .
     .
   END;
     .
     .
  END P';
```

We will say that the transformation T is valid if the program segments P and P' are equivalent (in the input-output sense) [ALLE71]. In case T is valid, we will say that the inner loop of P is reversible. We want to obtain (easily testable) sufficient conditions under which T is valid.

Let us denote the sequence of statements within the inner DO loop (of P) by F(I,J) where the subscripts indicate the value of the two loop control variables during the execution of the statement sequence F. Clearly, P and P' are equivalent, provided the following segments X and X' are equivalent.

X: F(I,1)
⋮
F(I,n_2)

X': F(I,n_2)
⋮
F(I,1)

A sufficient condition for X and X' to be equivalent is:

F(I,j) commutes with F(I,k) for

$\forall$ I $\in$ [1,n_1] and $\forall$ j, k $\in$ [1,n_2] j $\neq$ k.

Bernstein [BERN66] has derived sufficient conditions for commutivity of program segments. Note that, we need only pa ial commutivity here, i.e., we require that, F(I,j) F(I,k) sequence is equivalent to F(I,k) F(I,j) for $\forall$ k > j, $\forall$ I. We will repeat his conditions here. Let us denote the set of addresses fetched and not stored during the

execution of F(I,j) by W(I,j), the set of addresses stored and not fetched by X(I,j), the set of addresses fetched first and stored later by Y(I,j) and the set of addresses stored first and fetched later by Z(I,j). Let us denote by W_R, X_R, Y_R, Z_R, the similar set of addresses for the rest of the program segment P excluding the inner DO loop. Then, sufficient conditions for P and P' to be equivalent are:

1) $(W(I,k) \cup Y(I,k)) \cap (X(I,j) \cup Y(I,j) \cup Z(I,j)) = \emptyset$.

2) $(W(I,j) \cup Y(I,j)) \cap (X(I,k) \cup Y(I,k) \cup Z(I,k)) = \emptyset$.

3) $(X(I,k) \cup Y(I,k) \cup Z(I,k)) \cap$

$(X(I,j) \cup Y(I,j) \cup Z(I,j)) \cap (W_R \cup Y_R) = \emptyset$.

Note that, these are simple conditions which are easy to test. However, these conditions exclude the following case:

```
P1:  DO I = 1 TO n1;
       S = 0;
       DO J = 1 TO n2;
        S = S + A(I,J) * B(I,J);
       END;
        .
        .
        .
     END;
```

This case represents an inner product being calculated within the inner loop and as such is a most frequently occurring construct in matrix algorithms. For the program segment P_1, we have,

$W(I,J) = \{A(I,J), B(I,J)\}$,

$X(I,J) = \emptyset$, $Y(I,J) = \{S\}$,

$Z(I,J) = \emptyset$.

Therefore, $Y(I,J) \cap Y(I,k) = \{S\} \neq \emptyset$ which implies that conditions 1) and 2) are violated. But it is clear that the inner loop of P_1 is reversible. This shows that the conditions we have are sufficient but not necessary.

We will require that this construct be specially recognized as reversible. In fact, the following generalization is valid. Suppose, we have the following program segment:

```
DO I = 1 TO n1;
   :
   DO J = 1 TO n2;
     S = S □ E(J);
   END;
   :
END;
```

Assume that □ is a commutative and associative operator and E(J) is some expression, then the inner J loop is reversible.

Therefore, our loop reversal technique is a modified technique of Bernstein. Furthermore, it is generally profitable to reverse adjacent loops in the opposite directions. We apply this automated procedure to CD and see whether we can obtain CDR as the transformed procedure of section 4.1.

The two inner J loops are immediately recognized as reversible by the special construct. For the loop, 'DO I = 1 TO n-k', we have:

$$W(K,I) = \{A(K+I,1), \ldots A(K+I, K-1),$$
$$A(K,1) \quad \ldots A(K, k-1),$$
$$A(K,K)\},$$
$$X(K,I) = \emptyset,$$
$$Y(K,I) = \{A(K+I,K)\},$$
$$Z(K,I) = \{S\},$$
$$\Rightarrow W(K,I) \cup Y(K,I) = \{A(K+I,1{:}K),\ A(K,1{:}K)\}$$
$$X(K,J) \cup Y(K,J) \cup Z(K,J) = \{A(K+J,K),\ S\}.$$

Clearly, intersection is empty.

W_R = {A(K+1:n, 1:n-1)}

$X_R = \emptyset$, Y_R = {A(K+1, K+1), ..., A(n,n)}

Z_R = {S}.

The triple intersection is: $S \cap (W_R \cup Y_R) = \Phi$. Thus, the reversibility of the loop is established. By hand simulation, we have proved that our automated loop reversing procedure will indeed translate CD into CDR. In a similar fashion, we can show this for other algorithms.

We now turn our attention to programs written in OL/2. Quite clearly, the techniques presented for an element source language are applicable. We have, in addition, several more possibilities of reversal. Consider a vector assignment statement within a loop in an OL/2 program. The vector assignment statement will be compiled into a DO loop by the OL/2 compiler. If this loop is detected to be reversible, then the compiler can carry out the loop reversal. The conditions for reversibility are simpler in this case and are as follows (assume that the statement is a vector expression being assigned to a vector C):

1) Either the right side of the assignment statement does not contain the vector C, its ancestor, or subarrays in the ACB tree [PHIL72];

2) Or the assignment statement is given by C = C $\pm$ Exp where Exp is a vector expression satisfying condition 1.

If these conditions are satisfied then the vector assignment is reversible. Clearly, this procedure will translate OCD into CDR.

6.1.2 Loop Decomposition

We have not discussed this transformation in Chapter 4. Loop decomposition is also known as loop unswitching [ALLE71]. Consider the following program segment P:

```
P:  DO I = 1 TO n;
      A(I) = A(I) * A(I);
      B(I) = B(I) + I;

    END;
    rest of the program
    .
    .
    .
```

Now consider the following transformation P' of P:

```
P': DO I = 1 TO n;
      A(I) = A(I) * A(I);      P1
    END;
    DO I = 1 TO n;
      B(I) = B(I) + I;         P2
    END;
    .
    . rest of the program
    .
```

It is easily seen that the locality of P' is much better than that of P. In other words, whenever a single loop operates on two or more independent data streams then it is better to decouple these operations into two or more DO loops. Sufficient conditions for the transformation to be valid can be stated as follows [BERN66]:

a) $(W_1(I) \cup Y_1(I)) \cap (X_2(J) \cup Y_2(J) \cup Z_2(J)) = \emptyset,$

b) $(W_2(J) \cup Y_2(J)) \cap (X_1(I) \cup Y_1(I) \cup Z_1(I)) = \emptyset,$

c) $(X_1(I) \cup Z_1(I)) \cap (X_2(J) \cup Z_2(J)) \cap (W_R \cup Y_R) = \emptyset,$

for $1 \leq J < I \leq n.$

6.1.3 Submatrix-Multiplication-Transformation

In Chapter 4, many matrix algorithm contain matrix multiplication. It is, therefore, profitable to convert these internal matrix multiplications into submatrix multiplications.

In an array language like OL/2, the programmer is allowed to write a matrix multiplication as a primitive operation. In such a case, the compiler can easily compile this primitive into a submatrix multiplication. Thus OCD will be compiled into CDM. If we also apply the procedure of section 6.1.1 then, OCD will be compiled into CDMR with a corresponding high degree of locality. The same statement holds for other matrix algorithms.

For an element source language, it is more difficult since the compiler has to detect a sequence of statements which corresponds to matrix multiplication or its variants. A number of papers have appeared on the topic of automatic recognition of vector and matrix operations for the purpose of parallelism exploitation [SCHN72]. The same techniques are applicable for our purposes.

6.1.4 Conversion From a Non-Submatrix to a Submatrix Algorithm

In general, it is impossible to automate a conversion from a non-submatrix algorithm like CD to a submatrix algorithm like CDS. The transformation procedure would be required not only to understand the logic of the program, but also to have knowledge or 'intelligence' about the theory of matrix algebra. We will illustrate the difficulties involved in such a task by examples.

First we establish submatrix analogs (also known as block analogs) of scalar operations. For example, the submatrix analogs of scalar addition

and scalar multiplication are matrix addition and matrix multiplication respectively. When setting up the submatrix analog of division, we have a problem. Assume that in the non-submatrix version of a certain matrix program, the expression, E/A(k,k), occurs. A possible submatrix analog of this expression could be $E_S * (A_{I,I})^{-1}$, where $A_{I,I}$ is the (I,I) submatrix of A, I = ⌈K/m⌉, and E_S is the submatrix analog of E. Another possible analog of the expression is $(A_{I,I})^{-1} * E_S$. Clearly, these two expressions produce different effects, and only one of them is the correct analog. It is not at all clear from the original program which of the two possible analogs is correct. This particular example occurs in the transformation of CD into CDS (refer to section 4.1).

If we consider Cholesky decomposition, then the submatrix analog of the square root operation is the Cholesky decomposition of a submatrix. There is no way to automate this type of knowledge at present.

Another example occurs in attempting to transform ORTH into ORTHS (refer to section 4.2). The transformation is possible because of the known properties of orthonormal vectors [SCHO73,SCHW73]. Again, we cannot incorporate this knowledge in a compiler.

In case of eigenvalue problem, submatrix algorithms may not even exist. The basic problem is the noncommutative property of matrix multiplication [LOVA72].

6.2 Freeing and Prepaging

In this section, we will discuss the automation of freeing and prepaging for matrix algorithms, written in OL/2 language. It should be clear that, we can do the same kind of optimization for any other source

language. However, for an element source language the overhead incurred by our procedure is likely to be intolerable.

The optimizer will be a module of the OL/2 compiler. It will collect some information about the program during the syntactic phase and later use this information to do efficient freeing and prepaging for the program. Simply stated, the job of the optimizer consists of two parts: (a) prepage the subarrays and the arrays needed by the program at the proper time during the execution of the program and (b) free memory space occupied by the dead subarrays and dead arrays as soon as possible. We will use (sub) array to denote "subarray or array." Ideally, the proper time to prepage is such that any page of the (sub) array should be set up in MM before the first reference to it, and 'as soon as possible' is taken to mean 'just after the last use of the corresponding page'. Such an ideal situation will, clearly, be equivalent to the use of PWS paging algorithm and result in zero page faults as well as very economic use of memory. Such an ideal system, however, will be very expensive in terms of overhead. We consider a practical solution.

It is clear that some sort of a list of useful (sub) arrays needs to be prepared during the syntax analysis phase. This list can be used in two possible ways. First, it may be used to insert instructions for freeing and prepaging (sub) arrays during the coding phase. Second, it may be used dynamically, to prepage (sub) arrays from the list as far into future as demanded by the desired performance. Clearly, the second approach requires variable page allotment of MM. It will also incur more overhead since the list must be kept in MM during the execution

phase, but it is likely to result in an improved paging performance. We will discuss the implementation of the first approach, since it is more simple; however, the technique of constructing the list is common to both approaches.

Let us assume, for the sake of argument, that the program can acquire as many pages as necessary. Furthermore, assume that once a (sub) array has been brought into MM it remains there until after the last time it is used. Define the lifetime of a (sub) array A as follows: $L(A) = (t_1, t_2)$ where t_1 is the time of the first use and t_2 the time of the last use of A. Once the lifetime of each (sub) array is determined, then our problem is solved, at least in theory, since we can follow the following prepaging and freeing strategy:

Given $L(A) = (t_1, t_2)$, if $t < t_1 - T$ then PREPAGE(A); and if $t > t_2$ then FREE(A). The variable t is the current process time and T is the average page-fetch time.

We construct, therefore, the list USE_LIST whose nodes correspond to the (sub) arrays in the order in which they are referenced by the program. On the program level the normal sequential ordering is used, and within a statement, right end order of the expression tree is followed [JURI72]. The primary link in the USE_LIST is the one implied by the above ordering. We will refer to this link by the names, 'static link' and the NEXT pointer. In case of repeated use of an (sub) array, it is not worthwhile to enter each use in the USE_LIST, since the USE_LIST will become very long. Instead, we consider only the first and the last use and the interval between them has been defined as the lifetime. The decision made here is

quite arbitrary. We could have, alternately, defined the lifetime L' of (sub) array A as:

$$L'(A) = L(A) \cap SS(A)$$

where L(A) is the lifetime by the earlier definition and SS(A) is the syntactic scope of A (as in a block-structured language like OL/2). Note that, SS(A), in general, is a set of disjoint intervals and since L(A) is a single interval, L'(A) will be a set of disjoint intervals. The use of L' will involve more overhead but will result in obtaining better paging performance. There are many other ways of defining lifetime, but we will use the first definition, namely, L. We have to store L(A) for each A in the USE_LIST. A practical way to do this is to have a node for A corresponding to the first applied occurrence of A and store a lifetime pointer, called LP, in the node, pointing to the last applied occurrence of A.

We have tacitly assumed that the sequence of uses of (sub) arrays in a program can be linearized. There are several language constructs that prohibit this.

The GOTO statement introduces nonlinearity in the USE_LIST. Since the programming language that we are using (OL/2) is GOTO-less, we do not have this problem.

Another source of nonlinearity is the IF statement. Let us assume that the statement I = (IF E_1 THEN S_1 ELSE S_2) occurs in a program. Ideally, the USE_LIST for this statement should be:

$$U(I) = U(E_1) \left\langle \begin{matrix} T \\ F \end{matrix} \begin{matrix} U(S_1) \\ U(S_2) \end{matrix} \right\rangle$$

where U(X) denotes the USE_LIST for the program segment X. Since the branch actually taken by the program is unknown until the execution time, we have an obvious problem. The IF statement makes the predictability of the future reference pattern poor. Therefore, we have to settle for approximate solutions. There are various solutions to this problem.

The first solution is to linearize U(I) as $U(E_1) - U(S_1) - U(S_2)$. This solution implies that, independent of the branch taken by the program, the (sub) arrays in both S_1 and S_2 will be considered part of the data-working-set and will be prepaged. This has two implications: first, potentially useless pages are brought into MM incurring extra page pulls and wasting MM. Second, if $U(S_1)$ is large so that only part of it can be prepaged when the if-expression E_1 is evaluated and if the 'false' branch is taken then a good part of $U(S_2)$ will have to be demand paged.

The second approach assumes that $U(I) = U(E_1)$. In other words, we assume that $U(S_1) = U(S_2) = \text{null}$. This solution implies that depending on the branch taken by the program, $U(S_1)$ or $U(S_2)$ will have to be demand paged, but no memory space is wasted nor an extra channel traffic occurs.

The third approach allows the USE_LIST to be nonlinear and assumes that

$$U(I) = U(E_1) - \boxed{\text{STOP}} \begin{matrix} T \\ F \end{matrix} \left\langle \begin{matrix} U(S_1) \\ U(S_2) \end{matrix} \right\rangle .$$

This solution implies that all prepaging activity be stopped after the evaluation of the if-expression.

Which of these solutions is the most useful can only be decided by experience with a working system. We select the first solution for the rest of the discussion.

The existence of loops (e.g., FOR loops, WHILE loops) is still another source of nonlinearity. One possible way of linearizing, in this case, is to unfold the loops. But this solution is unattractive because: it is not applicable to WHILE loops since the number of times the loop is traversed is not known at compile time, and it is wasteful of memory space since a lot of information is duplicated in the USE_LIST. The alternative is to allow loops in the USE_LIST.

The fourth source of nonlinearity in the USE_LIST is the call to a procedure. We assume that each procedure has code inserted in it to do its own freeing and prepaging. Thus, in the calling procedure, we stop prepaging at and beyond the point of call and resume freeing and prepaging after a return from the called procedure. In our treatment, we have ignored call statements altogether but the above solution may easily be incorporated.

Assume that the USE_LIST has been constructed for a given program. Each (sub) array in the USE_LIST may have at most four boundaries. Now since OL/2 allows dynamic partitioning, these boundaries can move during the execution of the program. Therefore, each boundary has associated with it a boundary expression [PHIL72]. If all the boundary expressions of a (sub) array remain constant during the execution of a loop, then such a (sub) array will be called static (with respect to the loop). Those subarrays for which at least one of the boundary expressions vary within the loop are called dynamic subarrays. Clearly, paging techniques for these two types of (sub) arrays should be different. The static (sub) arrays need be prepaged only once. Whereas the extents of the dynamic subarrays change during the execution of the loop and hence parts of these subarrays

may have to be prepaged during the loop execution. We therefore, decide to originate a dynamic link at the beginning of a FOR loop and it will thread all subarrays within the scope of the FOR loop which are dynamic with respect to this loop. If the loop control variable is k, then the dynamic link will also be called k-chain. We define a predicate P such that for a subarray B and the loop control variable k, P(B,k) = T (true), iff B is dynamic with respect to loop k.

Once the USE_LIST has been constructed, as outlined above, prepaging instructions can be inserted during the coding phase as follows.

At nest level $\ell = 0$, we prepage (sub) arrays in the USE_LIST, traversing it along the static link. At nest level $\ell > 0$, prior to generating code for the loop, we insert instructions for prepaging (sub) arrays, traversing the USE_LIST along the static link; and within the loop, we insert instructions for prepaging dynamic subarrays traversing the USE_LIST along the dynamic link.

We will now discuss how to insert the lifetime pointers (LP) or equivalently, how to determine L(A) for each A in the USE_LIST. We assume the existence of a table of (sub) array names (addressed by the (sub) array name) and containing two pointers IP and CP for each entry. IP, the initial pointer, points to the first occurrence of the (sub) array in the USE_LIST and CP, the current pointer, points to a node in the USE_LIST, where currently last use of the (sub) array would have been listed if we were to list each use of the (sub) array. At the end of the syntax analysis phase, LP = CP for each (sub) array.

How and when to free (sub) arrays is the topic to be discussed next. A simple answer to this question is to free a (sub) array soon after its

end of lifetime. First of all, this implies the existence of a node, called the end-of-lifetime node, for each (sub) array in the USE_LIST. Whenever such a node is scanned during the traversal of USE_LIST in the coding phase, instructions can be inserted to free the corresponding (sub) array. However, there are several problems with this procedure. If the lifetime L(A) of a (sub) array A is completely contained within a loop f, then the above procedure will make us prepage and free A during every traversal of the loop. To alleviate this difficulty, we make the following change in our procedure. If at nest level $\ell = 0$ then we free A, on scanning the end-of-lifetime node of A; and if at nest level $\ell > 0$ then we free A, on scanning the end-of-lifetime node of A, only during the last traversal of the loop. Now if A is a dynamic subarray then this refined procedure also breaks down. For such a subarray, the contents of A during the last iteration of a loop may be significantly different from the contents of A during the previous iterations. The procedure we have outlined thus far, may leave a lot of dead elements in MM. A simple scheme immediately comes to mind to solve this problem. At the expiration of the lifetime of A within a loop, free only the dead elements of A, i.e., if we let A(k) to be the elements of A when the loop control variable takes the value k then at the expiration of lifetime of A during k^{th} loop traversal free (A(k+1) - A(k)). The danger in this procedure is that these freed elements may be the useful elements of an adjacent subarray during the next loop traversal. We solve this problem in the following way. Define the scope of an array A by,

$$S(A) = (\min_{B \subseteq A} IP(B), \max_{B \subseteq A} LP(B))$$

where $B \subseteq A$ means B is a subarray of A. Intuitively, S(A) is an interval specifying the first use and the last use of A or any of its subarrays. The scope S(A) of array A defines a natual universe over which we can define the dead subarrays of A. Define the DEAD_LIST as follows:

$$\text{DEAD\_LIST}(A, S(A)) = A - (A \cap \text{USE\_LIST}) - \{B \mid B \subseteq A,\ B \text{ is static with respect to all loops in which it is referenced}\}.$$

The expression $(A \cap \text{USE\_LIST})$ denotes the list of all subarrays of A that occur in the USE_LIST. Clearly, $B \in \text{DEAD\_LIST}$ implies that B is dynamic with respect to at least one loop. Now it is easy to merge the DEAD_LIST with the USE_LIST. For each array A, we do the following: for each $B \in \text{DEAD\_LIST}(A, S(A))$, and for each loop (with control variable k), in which B occurs, if $P(B, k) = T$, we link B to the dynamic chain of the loop. It is clearly, helpful to keep all dead subarrays at the head of the dynamic chain after which all the useful dynamic subarrays follow. The procedure of DEAD_LIST construction and linking to USE_LIST can only be done in a post-syntax, pre-coding phase. Now the code for freeing and prepaging dynamic subarrays is easily inserted in the loop, by traversing the dynamic chain of the loop.

Thus our description of the procedure to construct the USE_LIST is reasonably complete. However, we have ignored one important problem, namely, for $B \subset A$, $L(A) \cap L(B) \neq \emptyset$, i.e., the possibility that the lifetimes of a subarray B and (sub) array A may overlap. We solve this problem by considering several cases separately.

If $L(B) \subseteq L(A)$ then we delete B from the USE_LIST. Since we have assumed that the (sub) array A is resident in MM throughout its lifetime, therefore, its subarrays need not be considered separately.

If $L(B) \subseteq L(A)$ but $IP(B) < IP(A)$ and $LP(B) \leq LP(A)$ then we extend the lifetime of B so that $LP(B) = LP(A)$. The reason for this is that when the lifetime of B has ended, but the lifetime of A has not, B cannot be freed anyway.

If $LP(B) > LP(A)$ then we extend the lifetime of A so that $LP(A) = LP(B)$. If this simple approach is not used, then an alternative approach has to find the complement of B with respect to A and free all the the subarrays in the complement of B with respect to A and free all the subarrays in the complement. The procedure then becomes difficult and time consuming.

The next question to be discussed is, how to relate (sub) arrays to pages. The freeing and prepaging instructions available to us can free and prepage only pages, but the information in USE_LIST is about the (sub) arrays. Somehow, we have to bridge the gap. The problem is compounded by the fact that the subarrays can change their sizes during the execution of the program. We will consider the cases separately.

If an array occurs in the USE_LIST then we can prepage all of its pages a few nodes in advance. Similarly, at the expiration of its lifetime, we can free all of its pages. We have assumed that the array does not share any pages with other useful (sub) arrays.

If we consider a static subarray smaller than a page, then while prepaging it, we can just bring in the page in which the subarray is stored. While freeing, however, we have to make sure that this page does not contain any useful information. For the sake of simplicity, we will not free such a subarray.

If we consider a large static subarray then while prepaging, we fetch all the pages containing the elements of the subarray. While freeing, we can free only those pages which contain just the elements of this subarray. Other pages, which are shared with some other subarrays cannot be freed.

We now consider the case of dynamic subarrays. Consider a loop f with the control variable k such that for the given subarray B, $P(B,k) = T$ (i.e., B is a dynamic subarray with respect to the loop f). Let us denote the values that, the control variable k takes during the execution of the loop, by $k_0, k_1, \ldots, k_i, k_{i+1}, \ldots$. Let us denote the elements of subarray B, during the k^{th} loop traversal by B(k) and the pages containing B(k) by P(B(k)). Then $P(B(k_0)) \in DWS(k_0))$ and $\Delta P(B(k_{i+1})) = P(B(k_{i+1})) - P(B(k_i)) \in DWS(k_{i+1}) - DWS(k_i)$.

This means that the elements of $B(k_0)$ must be prepaged before the first traversal of the loop and that during each subsequent traversal $\Delta P(B(k_{i+1}))$ need be prepaged (if not null). The first requirement is already satisfied since before the first traversal of the loop, we prepage all the subarrays along the static chain in the loop.

For satisfying the second requirement, we must determine the conditions under which $\Delta P(B(k_{i+1}))$ will be non-null. Before that, however, we will determine the conditions under which $\Delta B(k_{i+1}) = B(k_{i+1}) - B(k_i)$ will be non-null. We will assume that the subarray B has four boundary expressions associated with it. Let us further assume that, B = A <I,J>, i.e., it is <I,J> subarray of A. Then r_{I-1}, r_I, C_{J-1}, C_J are the boundary expressions associated with it as shown in Figure 6.1.

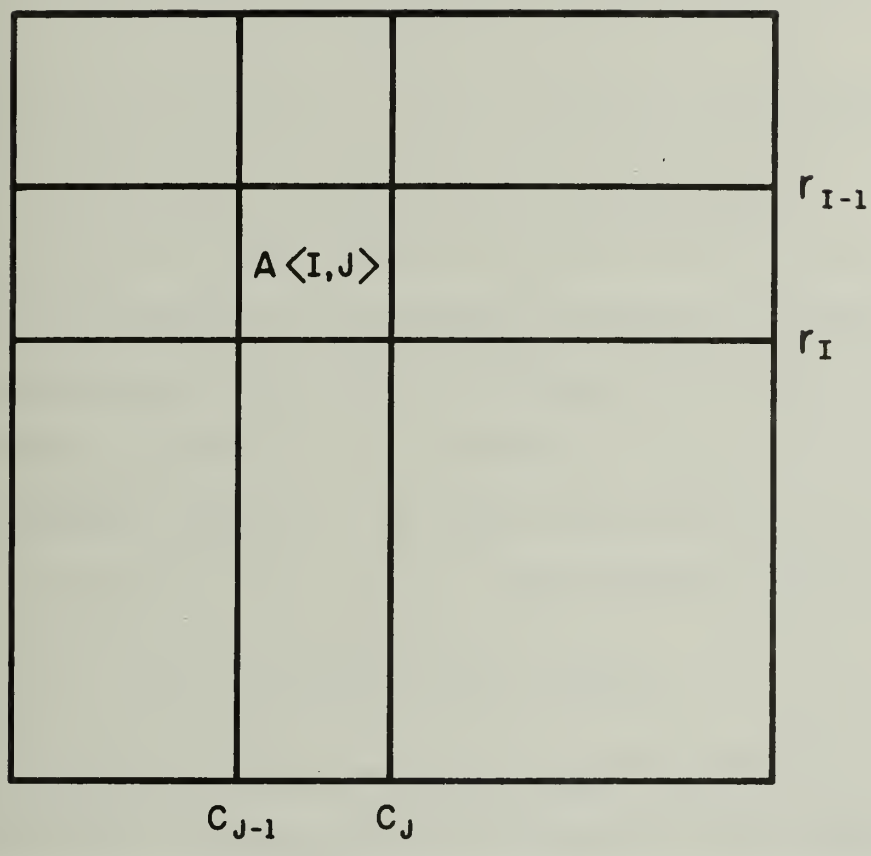


Figure 6.1. The Partition Lines

In general, these four expressions are functions of k. Let $X_1 = \{r_{I-1}(k), C_{J-1}(k)\}$, $X_2 = \{r_I(k), C_J(k)\}$ and $X = X_1 \cup X_2$. Define $f_k(x)\ \forall\ x \in X$ by:

$$f_{k_{i+1}}(x) = \begin{cases} 1 & \text{if } x(k_{i+1}) > x(k_i) \\ 0 & \text{if } x(k_{i+1}) = x(k_i) \\ -1 & \text{if } x(k_{i+1}) < x(k_i). \end{cases}$$

Then clearly, the condition for $\Delta\ B(k_{i+1})$ to be non-null is given by:

$$(\exists\ x \in X_1 \ni f_{k_{i+1}}(x) = -1) \vee (\exists\ x \in X_2 \ni f_{k_{i+1}}(x) = 1).$$

Intuitively, the condition asserts that the subarray B is expanding in at least one of the four directions. Let us assume that for a subset X_0 of X, $\forall\ x \in X_0 \Rightarrow$ B is expanding along the boundary expression x. We would like to determine the conditions under which $\Delta\ P(B(k_{i+1})) \neq \emptyset$. First of all, it is clear that,

$$\Delta\ B(k_{i+1}) = \emptyset \Rightarrow \Delta\ P(B(k_{i+1})) = \emptyset.$$

Given that $\Delta\ B(k_{i+1}) \neq \emptyset$, it may be that $\Delta\ P(B(k_{i+1})) = \emptyset$. If $\exists\ x \in X_0 \ni x(k_i) \equiv 0 \bmod m$, then $\Delta\ P(B(k_{i+1})) \neq \emptyset$. Intuitively, this condition asserts that one of the expanding boundaries has just crossed a page boundary. Thus the required condition for prepaging is:

$$(\Delta\ B(k_{i+1}) \neq \emptyset) \wedge (\exists\ x \in X_0 \ni x(k_i) \equiv 0 \bmod m).$$

As regards freeing dynamic subarrays, during the k^{th} loop traversal, we scan the dynamic chain and try to free all the dead subarrays. If the subarray shares pages with other subarrays then we cannot free these pages. Also if the subarray is contracting in at least one of the directions then we cannot free it, since the present elements of this dead subarray may become the future elements of some useful subarray. The condition under which a dead subarray B is not contracting in any direction is given by:

$$(\forall x \in X_1,\ f_{k_{i+1}}(x) \leq 0) \wedge (\forall x \in X_2,\ f_{k_{i+1}}(x) \geq 0).$$

The condition under which, the subarray B does not share pages with any other subarray is given by: $\forall x \in X,\ x(k_{i+1}) \equiv 0 \bmod m$. When both these conditions are satisfied, we can free all the pages containing the dead subarray B.

It may happen that some of the shared pages of a dead subarray belong to another dead subarray. This sharing will stop freeing of both these subarrays unnecessarily. We can easily incorporate a 'combination of dead subarray' function in our procedure. Assume that A <1,1>, A<1,2>, A<1,3> occur as dead subarrays and that there are only two vertical partition lines of A. Then the combination may be recognized as a whole (top) row of subarrays and may be denoted by A<1,*>. With this kind of combination, we may be able to free more frequently, in fact, such a situation does occur in Gaussian elimination. We will not, however, discuss the details.

The scheme for freeing that we have discussed is not complete, because there may exist a subarray which is dead but which always shared pages with useful subarrays, and therefore is never freed. To alleviate

this difficulty, when the scope S(A) of an array A ends, we will free all the subarrays of A remaining in MM. This of course, implies an end-of-scope node for each array A.

We give an algorithm (called ALG_AP), in Appendix D, which builds the USE_LIST, builds the DEAD_LIST and attaches to the USE_LIST and finally inserts instructions, in the code, for freeing and prepaging.

7. CONCLUSION AND FUTURE RESEARCH

7.0 Conclusion

The improvement of paging performance of programs running under a paged virtual memory system has been demonstrated in this thesis. As we have seen, there are many variables which affect the performance. The paging algorithm used by the system and the locality of the program under consideration are two of the more important variables of the system. We have studied the effects of these two variables on the performance of programs running under such a system.

In analyzing the performance, it was necessary to choose a set of performance measures. The traditional performance measure is the number of page pulls incurred by the program, but the number of page faults and the space-time product of main memory are also important performance measures. Which of these performance measures is the most important depends on a particular system and the mix of programs running under it.

Demand paging algorithms have been very popular, both in theory and in practice. We have shown, both theoretically and practically, that non-demand algorithms are useful in improving paging performance.

We have defined a fixed-memory demand prepaging algorithm DPMIN and proved that it is an optimal demand prepaging algorithm in the number of page faults. This algorithm, however, is unrealizable since it requires

the advance knowledge of the future reference string. It serves as a benchmark of performance for realizable demand prepaging algorithms.

We have defined several realizable prepaging algorithms. We have studied, for each of these algorithms, whether or not it is a stack algorithm. To show practical usefulness of these algorithms, we resort to a specific application. We have shown that these prepaging algorithms improve the paging performance of common matrix algorithms considerably. The reduction in the number of page faults is shown to be as high as an order of magnitude. We have measured the number of page faults and the number of page pulls for many different matrix algorithms using many different paging algorithms.

The application of realizable prepaging algorithms require some knowledge of the future paging needs of the program. If the programmer is willing to do this, he is suitably rewarded. We have shown that a compiler for a structured array-language like OL/2 can extract, without too much overhead, enough information from a program to improve its paging performance. The compiler can identify portions of the present working set which cease to be in the future working set. The compiler can also provide reliable advance information of the working set. As a result, moving unwanted information out of memory and bringing in useful information, in advance of its use, can be done efficiently.

We have defined a variable-memory prepaging algorithm PWS, based on Denning's WS algorithm. We have shown that PWS algorithm incurs zero page faults, has the same number of page pulls as WS, and it is more economical in ST product than WS. However, PWS is an unrealizable algorithm since it requires that the program's reference string be known in advance.

PWS algorithm can serve as a benchmark of performance among all variable-memory prepaging algorithms.

We have identified three methods of improving the locality of matrix algorithms. These are: loop reversal, submatrix multiplication and the construction of a submatrix algorithm. We have shown that these methods yield a vast improvement in locality and that the most powerful method is the construction of a submatrix algorithm. The average working set size is known to be a good measure of the locality of a program. We have shown that a decrease in the average working set size, generally, implies a reduction in the number of page faults.

We have considered the problem of automating these methods to improve locality. If the source language is a high-level array-language, like OL/2, then the nature of the primitives of the language provide a much easier means of optimization.

We have shown that the locality-improvement methods are effective for lower values of the page allotment and that prepaging methods are effective at higher values of the page allotment.

7.1 Future Research

As we have noted, very little is known about prepaging algorithms. Both, theoretical and practical investigations are needed for understanding the behavior of prepaging algorithms. In future, memory technology will provide very large memories making program (instruction) paging less important. Parallel processing will provide the means for solving problems with very large data bases, thereby making data paging more important. As a result we expect the emphasis in paging to shift toward specialized systems where more information can be used to anticipate paging demands.

There are many matrix algorithms, for which submatrix algorithms do not yet exist. This is a useful area of investigation.

Actual implementation, either in software or hardware, of the paging optimizer proposed in thesis needs to be done. The use of the paging optimizer in conjunction with variable-memory prepaging algorithms can also be investigated.

A study of models of program behavior in conjunction with prepaging algorithms is another fruitful area of investigation. In particular, we refer to conjecture 3.1 regarding the optimality of the paging algorithm DPOBLLRU for the sequential-random model of program behavior (refer to section 3.2.2.1).

LIST OF REFERENCES

[AHO71] Aho, A. V., Denning, P. J. and Ullman, J. D., "Principles of Optimal Page Replacement," JACM, Vol. 18, No. 1, Jan. 1971, pp. 80-93.

[ALLE71] Allen, F. E. and Cocke, J., "A Catalogue of Optimizing Transformations," IBM T. J. Watson Research Center Report RC-3548 (Sept. 1971).

[ANAC67] Anacker, W. and Wang, C. P., "Performance Evaluation of Computing Systems with Memory Hierarchies," IEEE Transactions on Computers, Vol. EC-16, No. 6, Dec. 1967, pp. 764-773.

[BAER71] Baer, J. L. and Sager, G. R., "Measurement and Improvements of Methods for Evaluation under Paging Systems," in Proc. of Conf. on Statistical Methods for Evaluation of Computer System Performance, Brown University, Nov. 1971, pp. 241-269.

[BAIR68] Bairstow, J. N., "Time-Sharing," Electronic Design, Vol. 16, No. 9 (1968), pp. C1-C22.

[BAYL68] Baylis, M. H. J., Fletcher, D. G. and Howarth, D. J., "Paging Studies Made on the I.C.T. Atlas Computer," IFIP (1968), D113.

[BELA66] Belady, L. A., "A Study of Replacement Algorithms for a Virtual Storage Computer," IBM Sys. J., Vol. 5, No. 2, (1966), pp. 78-101.

[BELA69] Belady, L. A. and Kuehner, C. J., "Dynamic Space Sharing in Computer Systems," CACM, Vol. 12, No. 5 (May 1969), pp. 282-288.

[BELA74] Belady, L. A. and Palermo, F. P., "On-line Measurement of Paging Behaviour by the Multivalued MIN Algorithm," IBM J.R.D. Jan. 1974, pp. 2-19.

[BERG71] Berge, C., Principles of Combinatorics, Academic Press, New York, 1971.

[BERN66] Bernstein, A. J., "Analysis of Programs for Parallel Programming," IEEE Transactions on Computers, Vol. EC-15, No. 5, (Oct. 1966), pp. 757-763.

[BOBR67] Bobrow, D. G. and Murphy, D. L., "Structure of a LISP System Using Two-Level Storage," CACM, Vol. 10, No. 3 (March 1967), p. 155.

[BRAW68] Brawn, B. and Gustavson, F., "Program Behaviour in a Paging Environment," AFIPS FJCC, Vol. 33 (1968), pp. 1019-1032.

[BRAW70] Brawn, B. and Gustavson, F., "Sorting in a Paging Environment," CACM, Vol. 13, No. 8 (Aug. 1970), p. 483.

[COFF68] Coffman, E. G. and Varian, L. C., "Further Experimental Data on the Behaviour of Programs in a Paging Environment," CACM, Vol. 11, No. 7 (July 1968), pp. 471-474.

[COFF72] Coffman, E. G. and Ryan, T., "A Study of Storage Partitioning Using a Mathematical Model of Locality," CACM, Vol. 15, No. 3 (Mar. 1972), pp. 185-190.

[COFF73] Coffman, E. G. and Denning, P. J., Operating System Theory, Prentice-Hall, Inc., Englewood Cliffs, N.J., 1973.

[COME67] Comeau, L. W., "A Study of the Effects of User Program Optimization in a Paging System," ACM Symposium on OS, Oct. 1967.

[CORB69] Corbato, F. J., "A Paging Experiment with the Multics System," Chapter 19, In Honor of P. M. Morse, M.I.T. Press, Cambridge, Massachusetts, 1967, pp. 217-228.

[DENN68] Denning, P. J., "Thrashing: Its Causes and Prevention," AFIPS SJCC, Vol. 33 (1968), pp. 915-922.

[DENN68a] Denning, P. J., "The Working Set Model of Program Behaviour," CACM, Vol. 11, No. 5 (1968), pp. 323-333.

[DENN68b] Denning, P. J., Chen, Y. C. and Shedler, G. S., "A Model for Program Behaviour under Demand Paging," Rep. RC-2301, IBM T. J. Watson Res. Center, Yorktown Heights, N. Y., Sept. 1968.

[DENN70] Denning, P. J., "Virtual Memory," Computing Surveys, Vol. 2, No. 3, Sept. 1970, pp. 153-189.

[DENN72] Denning, P. J., "On Modelling Program Behaviour," AFIPS SJCC, Vol. 40 (1972), pp. 937-944.

[DENN72a] Denning, P. J., Savage, J. E. and Spirn, J. R., "Models for Locality in Program Behaviour," TR-107, Dept. of Elect. Eng., Princeton Univ., Apr. 1972.

[DENN72b] Denning, P. J. and Schwartz, S. C., "Properties of the Working-Set Model," CACM, Vol. 15, No. 3 (Mar. 1972), pp. 191-198.

[DENN65] Dennis, J. B. and Glaser, E. L., "The Structure of On-Line Information Processing Systems," Proc. Second Congress on Information Systems Sciences, 1965, pp. 5-14.

[DOHE70] Doherty, W., "Scheduling TSS/360 for Responsiveness," AFIPS FJCC, Vol. 37 (1970), pp. 97-112.

[DOYL] Doyle, M. S., "The Effects of Program Behaviour on the Design of Storage Hierarchies," Ph.D. Thesis, University of Waterloo, Canada.

[DUBR72] Dubrulle, A. A., "Solution of the Complete Symmetric Eigenvalue Problem in a Virtual Memory Environment," IBM J.R.D., Nov. 1972, pp. 612-615.

[FERR73] Ferrari, D., "A Tool for Automatic Program Restructuring," Proc. ACM Nat. Conf., 1973, p. 228.

[FINE66] Fine, G. H., Jackson, C. W. and McIsaac, P. V., "Dynamic Program Behaviour Under Paging," Proc. 21st ACM Nat. Conf., 1966, pp. 223-228.

[FRAN74] Franaszek, P. A. and Wagner, T. J., "Some Distribution-free Aspects of Paging Algorithm Performance," JACM, Vol. 21, No. 1, (Jan. 1974), pp. 31-39.

[FREI68] Freibergs, I. F., "The Dynamic Behaviour of Programs," AFIPS FJCC, Vol. 33 (1968), pp. 1163-1168.

[GELE71] Gelenbe, E. and Tiberio, P., "The WSTI and Page Size in the Design of Virtual Memory Computer Systems," Proc. 9th Annual Allerton Conf. on Circuits and System Theory (1971), p. 104.

[GELE73] Gelenbe, E., Tiberio, P. and Boekhorst, J. C. A., "Page Size in Demand-Paging Systems," Proc. ACM SIGME Symp. (1973), pp. 1-12.

[GIBS66] Gibson, D. H., "Considerations in Block-Oriented Systems Design," AFIPS SJCC, Vol. 30 (1967), pp. 75-80.

[GORD73] Gordon, R. L., "The Organization and Control of a Slave Memory Hierarchy," TR-4429, Naval Underwater Systems Center, Feb. 1973.

[HATF71] Hatfield, D. J. and Gerald, J., "Program Restructuring for Virtual Memory," IBM Sys. J., Vol. 10, No. 3 (1971), pp. 168-192.

[HATF72] Hatfield, D. J., "Experiments on Page Size, Program Access Patterns and Virtual Memory Performance," IBM J.R.D., Jan. 1972, p. 58.

[HOAR72] Hoare, C. A. R., "A Survey of Store Management Techniques Part Two," in Operating Systems Techniques, (Ed.) Hoare and Perrott, APIC Studies in Data Processing, No. 9, 1972.

[HOUS64] Householder, A. S., The Theory of Matrices in Numerical Analysis, Blaisdell Publishing Company, New York, 1964.

[ISAA66] Isaacson, E. and Keller, H. B., Analysis of Numerical Methods, John Wiley and Sons, 1966.

[JOSE70] Joseph, M., "An Analysis of Paging and Program Behaviour," The Computer Journal, Vol. 13, No. 1, Feb. 1970, p. 48.

[JURI72] Jurich, D. R., "An Approach to the Compilation of Array Expressions in the OL/2 Language," University of Illinois at Urbana-Champaign, Department of Computer Science Report No. 550, 1972.

[KILB62] Kilburn, T., Edwards, D. G., Lanigan, M. J. and Sumner, F. H., "One Level Storage System," IRE, EC-11, No. 2 (Apr. 1962), pp. 223-235.

[KING71] King, W. F., "Analysis of Demand Paging Algorithms," IFIP 1971, Vol. 1, p. 485.

[KNIG] Knight, J. C. and Page, E. S., "Searching on a Paging Environment," to appear.

[KUCK70] Kuck, D. J. and Lawrie, D. H., "The Use and Performance of Memory Hierarchies: A Survey," Software Engineering, Vol. 1, 1970, pp. 45-77.

[KUEH68] Kuehner, C. and Randell, B., "Demand Paging in Perspective," AFIPS SJCC, Vol. 32 (1968), pp. 1011-1018.

[LEHM68] Lehman, M. M. and Rosenfeld, J. L., "Performance of a Simulated Multiprogramming System," AFIPS SJCC, Vol. 33 (1968), pp. 1431-1442.

[LIPT68] Liptay, J. S., "Structural Aspects of the System 360/85: II, The Cache," IBM Sys. J., Vol. 7, No. 1 (1968), p. 15.

[LOVA72] Lovass-Nagy, V. and Powers, D. L., "A Note on Block Diagonalization of Some Partitioned Matrices," Linear Algebra and Its Applications, Vol. 5, Oct. 1972, pp. 339-346.

[MASU72] Masuda, T., Takahashi, N. and Yoshizawa, Y., "The Comparison of Swapping Algorithms and Some Program Behaviour Under a Paging Environment," Information Processing in Japan, Vol. 12 (1972), pp. 70-75.

[MATT70] Mattson, R. L., Gecsei, J., Slutz, D. R. and Traiger, I. L., "Evaluation Techniques of Storage Hierarchies," IBM Sys. J., Vol. 9, No. 2 (1970), pp. 78-117.

[MCKE69] McKeller, A. C. and Coffman, E. G., "The Organization of Matrices and Matrix Operations in a Paged Multiprogramming Environment, CACM, Vol. 12, No. 3 (1969), pp. 153-165.

[MOLE72] Moler, C. B., "Matrix Computations with Fortran and Paging," CACM, Vol. 15, No. 4 (1972), p. 268.

[ONEI67] O'Neill, R. W., "Experience Using a Time Sharing Multiprogramming System with Dynamic Address Relocation Hardware," AFIPS SJCC, Vol. 30 (1967), pp. 611-621.

[OPPE68] Oppenheimer, G. and Weizer, N., "Resource Management for a Medium Scale Time Sharing Operating System," CACM, Vol. 11, No. 5, (May 1968), p. 113.

[ORGA72] Organick, E. I., The MULTICS System: An Examination of Its Structure, MIT Press, Cambridge, Mass., 1972.

[PHIL72] Phillips, J. R. and Adams, H. C., "Dynamic Partitioning for Array Languages," CACM, Vol. 15, No. 12 (Dec. 1972), pp. 1023-1032.

[PINK68] Pinkerton, T., "Program Behaviour and Control in Virtual Storage Computer Systems," University of Michigan, CONCOMP Report 4 (Apr. 1968).

[POME71] Pomeranz, J. E., "Paging with Fewest Expected Replacements," IFIP, (1971), Vol. 1, pp. 491-493.

[RAMA66] Ramamoorthy, C. V., "The Analytical Design of a Dynamic Lookahead and Program Segmenting Scheme for Multiprogrammed Computers," Proc. 21st ACM Nat. Conf., 1966, pp. 229-239.

[RAND68] Randell, B. and Kuehner, C. J., "Dynamic Storage Allocation Systems," CACM, Vol. 11, No. 5 (May 1968), pp. 297-306.

[RAND69] Randell, B., "A Note on Storage Fragmentation and Program Segmentation," CACM, Vol. 12, No. 7 (July 1969), p. 365.

[RODR72] Rodriguez-Rosell, J. and Dupuy, J. P., "The Evaluation of a Time-Sharing, Page Demand System," AFIPS SJCC (1972), Vol. 40, pp. 759-765.

[RODR73] Rodriguez-Rosell, J. and Dupuy, J. P., "The Design, Implementation, and Evaluation of a Working Set Dispatcher," CACM, Vol. 16, No. 4 (Apr. 1973), pp. 247-253.

[ROGE73] Rogers, L. D., "Optimal Paging Strategies and Stability Considerations for Solving Large Linear Systems," Ph.D. Thesis University of Waterloo, Canada (1973).

[SALT74] Saltzer, J. H., "A Simple Linear Model of Demand Paging Performance," CACM, Vol. 17, No. 4 (Apr. 1974), pp. 181-186.

[SAYR69] Sayre, D., "Is Automatic Folding of Programs Efficient Enough to Displace Manual?" CACM Vol. 12, No. 12 (Dec. 1969), pp. 656-660.

[SCHN72] Schneck, P. B., "Automatic Recognition of Vector and Parallel Operations in a Higher Level Language," Proc. ACM Nat. Conf. (Aug. 1972), pp. 772-780.

[SCHO73] Schonage, A., "Fast Schmidt Ortogonalization and Unitary Transformations of Large Matrices," in Complexity of Sequential and Parallel Numerical Algorithms, Ed. J. F. Traub, Proc. Symp. (1973), Carnegie-Mellon University.

[SCHW73] Schwarz, H. R., Rutishauser, H. and Stiefel, E., Numerical Analysis of Symmetric Matrices, Prentice-Hall, Inc., Englewood Cliffs, N.J. (1973), translated by Hertelendy, P.

[SHED72] Shedler, G. S. and Tung, C., "Locality in Page Reference Strings," SIAM J. on Comput., Vol. 1, No. 3 (Sept. 1972), pp. 218-241.

[SHEM66] Shemer, J. E. and Shippey, G. A., "Statistical Analysis of Paged and Segmented Computer Systems," IEEE, Vol. EC-15, No. 6 (Dec. 1966), pp. 855-863.

[SHEM69] Shemer, J. E. and Gupta, S. C., "On the Design of Bayesian Storage Allocation Algorithms for Paging and Segmentation," IEEE, Vol. C-18, No. 7 (July 1969), pp. 644-651.

[SISS68] Sisson, S. S. and Flynn, M., "Addressing Patterns and Memory Handling Algorithms," AFIPS FJCC, Vol. 33, No. 2, (1968), pp. 957-967.

[SMIT67] Smith, J. L., "Multiprogramming under a Page on Demand Strategy," CACM, Vol. 10, No. 10 (Oct. 1967), pp. 636-646.

[STEV68] Stevenson, D. A. and Vermillion, W. H., "Core Storage as a Slave Memory for Disk Storage Devices," IFIP (1968), pp. F86-F91.

[VARI68] Varian, L. C. and Coffman, E. G., "An Empirical Study of the Behaviour of Programs in a Paging Environment," CACM, Vol. 11, No. 7 (July 1968), pp. 471-474.

[WEIN72] Weinberg, G. M., "Programming and Compiling Strategies for Paging Systems," Software-Practice and Experience, Vol. 2, 1972, pp. 165-171.

[WEIZ69] Weizer, N. and Oppenheimer, G., "Virtual Memory Management in a Paging Environment," AFIPS SJCC (1969), Vol. 34, pp. 270-271.

[WILK63] Wilkinson, J. H., Rounding Errors in Algebraic Processes, Her Majesty's Stationery Office, London (1963).

[WILK71] Wilkinson, J. H. and Reinsch, C., Linear Algebra, Springer Verlag, 1971.

[YELO71] Yelowitz, L., "Toward Optimal Paginations for Structured Programs," Symposium on Computers and Automata, Polytechnic Institute of Brooklyn, 1971, pp. 225-236.

APPENDIX A

PL/I PROGRAMS FOR THE SIMULATION OF SOME PAGING ALGORITHMS

A.1 LRU_WS_MIN

We will present LRU_WS_MIN in this section. There are three modules: the first module declares all the global variables used by LRU_WS_MIN, the second module is the procedure LRU_WS_MIN and the third module computes the results. Note that LRU_WS_MIN is a stack algorithm and therefore, in one pass of the reference string, obtains the page fault behavior for all possible values of the page allotment or the window size whichever is pertinent to the algorithm.

```
MODULE1:/*DECLARE ALL GLOBAL VARIABLES                          */
DCL(NA                          , /*ADDRESS SPACE SIZE  */
    LRUCNT(0:NA)  INIT((NA+1)(0)), /*LRU DIST.COUNTER    */
    WSCNT (0:NA)  INIT((NA+1)(0)), /*REF.INT.FREQ.COUNTER*/
    MINCNT(0:NA)  INIT((NA+1)(0)), /*MMC FREQ.COUNTER    */
    X(NA+1)       INIT((NA+1)(0)), /*LRUSTACKPAGEXINDEX  */
    CURSTACK      INIT(0)        , /*LRUSTACK POINTER    */
    LNGTH         INIT(0)        , /*REF.STR. LENGTH     */
    C                            , /*MEMORY SIZE         */
    NCLRU                        , /*LRU SUCC.FUNCTION   */
    NTWS                         , /*WS  SUCC.FUNCTION   */
    NCMIN                        , /*MIN SUCC.FUNCTION.  */
    PCLRU,PTWS,PCMIN,              /*THEIR PAGE FAULTS   */
    REFCNT(NA+1)  INIT((NA+1)(0)), /*REF.INTERVAL COUNTER*/
    P(NA+1)       INIT((NA+1)(0)), /*P -STACK            */
    P_POINT       INIT(0)        , /*P_STACK POINTER     */
    #) FIXED BIN(31,0);
DCL(MCLRU,MTWS,MCMIN,              /*PAGE FAULT FREQENCIE*/
    STAU             ,             /*AVE.WS SIZE         */
    ##) FLOAT BIN;
MODULE2:/* PROCEDURE LRU_WS_MIN                                */
LRU_WS_MIN:PROC(XX);
/*THIS PROC SIMULATES MATTSON-LRU-STACK-ALG.TOGETHER*/
/*WITH GORDON-MODIFICATION TO GET WS-STATISTICS AND */
/*ONE PASS MIN CF.BELADY & PALERMO                  */
 DCL(XX          , /*CURRENTLY REFERENCED PAGE */
     DIST INIT(0), /*STACK DISTANCE*/
     INT  INIT(0), /*REF. INTERVAL */
     I,J,JP,MINP,MINPP) FIXED BIN(31,0);
 LRU_WS: /*SEE R.L.GORDON */
 LNGTH=LNGTH+1;CURSTACK=CURSTACK+1;
 /*PUT THIS PAGE ON TOP OF STACK  */
 X(CURSTACK)=XX;REFCNT(CURSTACK)=1;
 /*SEARCH FOR THIS PAGE IN THE STACK   */
 DO I=CURSTACK-1 TO 1 BY -1;
  IF X(I)=XX  THEN
   DO; /* FOUND */
    INT=REFCNT(I);DIST=CURSTACK-I;
    /*MOVE PART OF STACK ONE BELOW */
    DO J=I TO CURSTACK-1 BY 1;
     X(J)=X(J+1);REFCNT(J)=REFCNT(J+1);
    END;
   END;ELSE REFCNT(I)=REFCNT(I)+1;
 END;
 /* STEP LRU & WS COUNTERS   */
 IF INT>NA THEN INT=NA;/*DONT TRUST WSCNT(NA)*/
 LRUCNT(DIST)=LRUCNT(DIST)+1;WSCNT(INT)=WSCNT(INT)+1;
 IF DIST=0 THEN DIST=CURSTACK-1;ELSE
 DO;CURSTACK=CURSTACK-1;DIST=DIST-1; END;
 /*NEXT IS THE CODE FOR BELADYS M-OPERATOR */
```

```
 MIN:
 IF DIST=0 THEN RETURN;
 IF DIST>P_POINT THEN
 DO; /* A FRESH PAGE-REF. */
  P_POINT=P_POINT+1;P(P_POINT)=DIST+1;RETURN;
 END;
 /*FIND SMALLEST ELE BELOW DIST IN P-STACK */
 MINP=P(1);J=1;LSMALL=P_POINT-DIST+1;
 DO I=2 TO LSMALL BY 1;
  IF P(I) < MINP THEN
  DO;MINP=P(I);J=I; END;
 END;
 DO WHILE('1'B);
  IF J=LSMALL THEN
  DO;
   DO I=J TO P_POINT-1 BY 1;
    P(I)=P(I+1);
   END;
   P(P_POINT)=MINP;MINCNT(MINP)=MINCNT(MINP)+1;
   RETURN;
  END;
  MINPP=P(J+1);JP=J+1;
  DO I=J+2 TO LSMALL BY 1;
   IF P(I) < MINPP THEN
   DO;MINPP=P(I);JP=I;END;
  END;
  P(JP)=MINP;P(J)=MINPP;J=JP;
 END;
END LRU_WS_MIN;
```

A.2 DPMIN

In this section, we will present DPMIN. There are three modules: the first module declares all the global variables used by DPMIN, the second module is the procedure DPMIN and the third module computes and prints the required statistics. Note that, DPMIN is not a stack algorithm and therefore, requires one pass of the reference string for each value of the page allotment c.

```
MODULE1:/*DECLARE ALL GLOBAL VARIABLES                          */
DCL(NA                          ,/*SIZE OF THE ADDRESS SPACE */
    DM_STK(2,NA) INIT((2*NA)(0)),/*TWO COL. OF NUM.MATRIX  */
    PREV         INIT(1)        ,/*INDICATES LAST MEMSTATE */
    CURR         INIT(2)        ,/*INDICATES CURR MEMSTATE */
    CURR_CNT     INIT(0)        ,/**#PAGES IN MEMSTATE      */
    FULL         INIT(1)        ,
    EMPTY        INIT(0)        ,
    LNGTH        INIT(0)        ,
    C                           ,/**#OF MEM.PAGES ALLOCATED */
    FAULTS       INIT(0)        ,/**#OF PAGE FAULTS */
    PULLS        INIT(0)        ,/**#OF PAGE PULLS  */
                 ) FIXED BIN(31,0);
MODULE2:/*PROCEDURE DPMIN                                       */
DPMIN:PROC(X);
/*THIS PROC.OBTAINS STAT. FOR DPMIN  */
 DCL(X   ,       /*PAGE     INDEX    */
     I,ITEMP) FIXED BIN(31,0);
 LNGTH=LNGTH+1;
 IF CURR_CNT = C THEN
 DO; /*PRESENT MEMSTATE FULL */
  /*IF THIS PAGE IN MEM.THEN QUIT */
  IF DM_STK(CURR,X)=FULL THEN RETURN;
  ITEMP=PREV;PREV=CURR;CURR=ITEMP;
  DO I=1 TO NA;
   DM_STK(CURR,I)=EMPTY;
  END;
  DM_STK(CURR,X)=FULL;CURR_CNT=1;FAULTS=FAULTS+1;
  RETURN;
 END;
IF DM_STK(CURR,X)=FULL THEN RETURN;
IF DM_STK(PREV,X)=FULL THEN
 DO;DM_STK(CURR,X)=FULL;CURR_CNT=CURR_CNT+1;RETURN;END;
DM_STK(CURR,X)=FULL;CURR_CNT=CURR_CNT+1;
IF CURR_CNT=1 THEN FAULTS=FAULTS+1;
ELSE               PULLS =PULLS +1;
END DPMIN;
MODULE3:/*COMPUTATION OF RESULTS                                */
 PULLS=FAULTS+PULLS;
 PUT SKIP DATA(C,FAULTS,PULLS,LNGTH);
```

A.3 FREELRU

In this section, we will present FREELRU. This is a stack algorithm and therefore, it obtains the page fault statistics, for all possible values of the page allotment c, in one pass of the reference string. There are three modules: the first module declares all the global variables used by FREELRU, the second module is the procedure FREELRU and the third module computes the results.

```
MODULE1:/*DECLARE ALL GLOBAL VARIABLES    */
DCL(NA                              ,/*SIZE OF ADDRESS SPACE */
    DISTCNT(0:NA) INIT((NA+1)(0)),/*LRU DIST.COUNTER       */
    X(NA+1)       INIT((NA+1)(0)),/*STACK PAGE INDEX      */
    PSTATE(NA+1)  INIT((NA+1)(0)),/*EMPTY-FULL INDICATOR  */
    EMPTY         INIT(0),
    FULL          INIT(1),
    NORMAL        INIT(0),        /*INDICATES A PAGE REF. */
    DFREE         INIT(1),        /*INDICATES PAGE IS DEAD*/
    CURSTACK      INIT(0),        /*POINTER TO STACK     */
    LNGTH         INIT(0),        /*REF. STRONG LENGTH   */
    C                    ,        /*MEM.PAGE ALLOTMENT   */
    NC                   ,        /*SUCCESS FUNCTION     */
    FAULTS        INIT(0),        /**OF PAGEFAULTS      */
    #) FIXED BIN(31,0);
DCL(FC ,/*SUCCESSFREQENCY*/
    MC ,/*PAGEFAULT FREQ.*/
    ##) FLOAT BIN;
MODULE2:/*PROC. FREELRU(STACK ALGORITHM) */
FREELRU:PROC(ACTION,XX);
 DCL(ACTION,/*INDICATES WHETHER A PAGE REF. OR A FREE COMM*/
     XX    ,/*CURRETLY REFERENCED PAGE */
     I,J,K,PTEMP,XTEMP,TEMP,FLAG) FIXED BIN(31,0);
 IF ACTION=NORMAL THEN
  DO;/*A REFERENCE TO PAGE XX*/
   LNGTH=LNGTH + 1;
        /*SEARCH FOR THIS PAGE IN STACK  */
   FLAG=0;
   DO I=1 TO CURSTACK WHILE(FLAG=0);
    IF PSTATE(I)=FULL & X(I)=XX THEN
     DO;
      FLAG=1; /* TO TERMINATE SEARCH */
      PSTATE(I)=EMPTY;
      DISTCNT(I)=DISTCNT(I)+1;
     END;
   END;
   IF FLAG=0 THEN
   /* PAGE NOT FOUND IN CORE,DISTCNT(0) IS ABS. */
   /*PAGE FAULT COUNTER */
    DISTCNT(0)=DISTCNT(0)+1;
   /* PUT CURR PAGE ON STACK */
   PTEMP=PSTATE(1);XTEMP=X(1);PSTATE(1)=FULL;
   X(1)=XX;K=1;
   DO J=2 TO CURSTACK+1 WHILE(PTEMP=FULL);
    TEMP=PSTATE(J);PSTATE(J)=PTEMP;PTEMP=TEMP;
    TEMP=X(J);X(J)=XTEMP;XTEMP=TEMP;K=J;
   END;
   IF K>CURSTACK THEN CURSTACK=K;
  END;/*CODE FOR ACTION=NORMAL ENDS */
  IF ACTION=DFREE THEN
```

```
   /*DECLARE PAGE XX DEAD IF IN MEM. */
   DO;
    FLAG=0;
    DO I=1 TO CURSTACK WHILE(FLAG=0);
     IF PSTATE(I)=FULL & X(I)=XX THEN
      DO;FLAG=1;PSTATE(I)=EMPTY;K=I;END;
    END;
    IF FLAG=1 THEN IF K=CURSTACK THEN
                      CURSTACK=CURSTACK-1;
END FREELRU;
MODULE3:/*RESULT COMPUTATION */
NC=0;
DO C=1 TO NA;
 NC=NC+DISTCNT(C);FC=NC/FLOAT(LNGTH);
 MC=1.0-FC;FAULTS=LNGTH-NC;
 PUT SKIP DATA(C,NC,FAULTS,FC,MC);
END;
```

A.4 FREEDPRE4LRU

In this section, we will present FREEDPRE4LRU. This is not a stack algorithm and therefore, requires one pass of the reference string for each value of the page allotment c. There are three modules: the first module declares all the global variables, the second module is the procedure FREEDPRE4LRU and the third module computes and prints out the required statistics.

```
MODULE1:/*DECLARE ALL GLOBAL VARIABLES          */
DCL(NA             ,/*SIZE OF ADDRESS SPACE     */
    DP4_PT(NA)     INIT((NA)(0)), /*PAGETABLE  */
    DP4_STACK(NA) INIT((NA)(0)), /*LRU STACK  */
    NORMAL         INIT(0)      , /*PAGE REF.  */
    PREPAGE        INIT(1)      ,
    DFREE          INIT(2)      ,
    DP4_POINT      INIT(0)      , /*LRUSTACK POINTER */
    FAULTS         INIT(0)      , /*#OF PEGE FAULTS  */
    PULLS          INIT(0)      , /*#OF PAGE PULLS   */
    PRECNT         INIT(0)      , /*#OF PREPAGED PAGES IN MEM */
    LNGTH          INIT(0)      , /*REF.STRING LNGTH*/
    C, /*MEM.SIZE */
    #) FIXED BIN(31,0);
MODULE2:/*PROC FREEDPRE4LRU                     */
FREEDPRE4LRU:PROC(ACTION,X);
/* THIS PROC.OBTAINS #PAGE FAULTS & #PAGE PULLS     */
/*FOR FREEDPRE4LRU PAGING ALGORITHM                 */
 DCL(ACTION, /*INDICATES WHETHER TO FREE,PREEPAGE OR*/
             /*A REFERENCE TO PAGE X                */
     X,      /*CURRENTLY REFERENCED PAGE            */
 DCL(NOTPRESENT  INIT(0),
     NOTSETUP    INIT(1),
     NOTUSED     INIT(2),
     USED        INIT(3),
     I,J) FIXED BIN(31,0);
 DCL(LACTION(0:2) INIT(LNORMAL,LPREPAGE,LFREE),
     LPAGE  (0:3) INIT(NOT_PRESENT,NOT_SETUP,
                       NOT_USED,LUSED))LABEL;
 GOTO LACTION(ACTION);
 /*A REFERENCE TO PAGE X */
 LNORMAL:LNGTH=LNGTH+1;
  GOTO LPAGE(DP4_PT(X));
  NOT_SETUP:/*REQUIRED PAGE NOT YET SETUP IN MEM */
   FAULTS=FAULTS+1;/* INC. PAGEFAULT COUNT */
   /*PUT THIS PAGE ON STACK */
   DP4_POINT=DP4_POINT+1;
   DP4_STACK(DP4_POINT)=X;
   DP4_PT(X)=USED;PRECNT=PRECNT-1;
   /*DECLARE ALL NOTSETUP PAGES SETUP */
   DO I=1 TO NA;
    IF DP4_PT(I)=NOTSETUP THEN
    DO;DP4_PT(I)=NOTUSED;PULLS=PULLS+1;END;
   END;
  RETURN;
  NOT_USED:/*REQ.PAGE SETUP */
   DP4_PT(X)=USED;PRECNT=PRECNT-1;
   DP4_POINT=DP4_POINT+1;DP4_STACK(DP4_POINT)=X;
  RETURN;
  LUSED:/*PAGE X HAS BEEN USED BEFORE  */
   DO I=DP4_POINT TO 1 BY -1;
    IF DP4_STACK(I)=X THEN
    DO;
     DO J=I TO DP4_POINT-1;
      DP4_STACK(J)=DP4_STACK(J+1);
     END;
     DP4_STACK(DP4_POINT)=X;RETURN;
    END;
   END;
  /*PAGE X IS NOT IN MEM    */
  NOT_PRESENT:FAULTS=FAULTS+1;
```

```
   IF DP4_POINT+PRECNT >=C THEN
   DO;
    DP4_PT(DP4_STACK(1))=NOTPRESENT ;/*REPLACED PAGE */
    DO J=1 TO DP4_POINT-1;
     DP4_STACK(J)=DP4_STACK(J+1);
    END;
   END;
   ELSE DP4_POINT=DP4_POINT+1;
   DP4_STACK(DP4_POINT)=X;
   DP4_PT(X)=USED;
   /*DECLARE ALL NOTSETJP PAGES SETUP */
   DO I=1 TO NA;
    IF DP4_PT(I)=NOTSETUP THEN
    DO;DP4_PT(I)=NOTUSED;PULLS=PULLS+1;END;
   END;
  RETURN;
 LPREPAGE:/*PREPAGE PAGE X    */
  IF DP4_PT(X)=NOTPRESENT & DP4_POINT+PRECNT<C & PRECNT < C-1 THEN
  DO;DP4_PT(X)=NOTSETUP;PRECNT=PRECNT+1;END;
 RETURN;
 LFREE:/*PAGE  X IS KNOWN TO BE DEAD NOW */
  IF DP4_PT(X) =USED THEN
  DO I=DP4_POINT TO 1 BY -1;
   IF DP4_STACK(I)=X THEN
   DO;
   DO J= I TO DP4_POINT-1;
    DP4_STACK(J)=DP4_STACK(J+1);
   END;
   DP4_POINT=DP4_POINT-1;
   END;
  END;
  DP4_PT(X)=NOTPRESENT;
 RETURN;
END FREEDPRE4LRU;
MODULE3:/*COMPUTATION OF RESULT       */
 PULLS=FAULTS+PULLS;
 PUT SKIP DATA(C,FAULTS,PULLS,LNGTH);
```

APPENDIX B

PROOF OF THEOREM 3.4

Theorem 3.4: PRE3A is a stack algorithm for $\{\omega | \omega \in N^+, P(\omega)\}$.

Proof: It is sufficient to prove that

$$S_t(c) \subseteq S_t(c+1) \qquad \forall\, c \geq 1, \forall\, t \geq 0.$$

The proof is by induction on t. Assume inductively that,

(1) $U_t(c+1) = U_t(c) + y_1$ where $y_1 = \emptyset$ or $y_1 \in N$.

(2) $N_t(c+1) = N_t(c) + y_2$ where $y_2 = \emptyset$ or $y_2 \in N$.

(3) $S_t(c+1) = S_t(c) + y_3$ where $y_3 = \emptyset$ or $y_3 = y_1$ or $y_3 = y_2$.

(4) $\text{COUNT}(y, t, c) = \text{COUNT}(y, t, c+1)$, $\forall\, y \in N_t(c)$.

Clearly, all these assumptions are trivially true for t = 0. Now depending on r_{t+1} there are two cases to consider:

case (1) $r_{t+1} = x \in N$. Now depending on x, there are four case to consider:

1(a): $x \in N_t(c+1)$ which implies $x \in N_t(c)$ or $x \notin N_t(c)$.

Assume first that $x \in N_t(c)$.

From step 1(a) of PRE3A,

$$\forall\, y \in N_t(c),\ \text{COUNT}(y, t+1, c) = \text{COUNT}(y, t, c) + 1$$
$$= \text{COUNT}(y, t, c+1) + 1$$
$$= \text{COUNT}(y, t+1, c+1).$$

Let $N_t(c+1) = N_t(c) + y_2$.

Then

$$\{y|y \in N_t(c+1),\ \mathrm{COUNT}(y,\ t+1,\ c+1) \geq T\}$$

$$= \{y|y \in N_t(c),\ \mathrm{COUNT}(y,\ t+1,\ c+1) \geq T\} +$$

$$\text{if } \mathrm{COUNT}(y_1,\ t+1,\ c+1) \geq T \text{ then } y_2$$

$$= \{y|y \in N_t(c),\ \mathrm{COUNT}(y,\ t+1,\ c) \geq T\} +$$

$$\text{if } \mathrm{COUNT}(y_1,\ t+1,\ c+1) \geq T \text{ then } y_2.$$

Also let $U_t(c+1) = U_t(c) + y_1$ where y_1 and y_2 cannot both be nonzero.

Therefore, we have, from step 1(a) of PRE3A,

$$U_{t+1}(c+1) = U_{t+1}(c) + (y_1 \text{ or } y_2 \text{ or null})$$

$$N_{t+1}(c+1) = N_{t+1}(c) + (y_2 \text{ or null})$$

This proves all the inductive steps.

Next assume $x \in N_t(c)$. Now if $N_t(c+1) = N_t(c) + y_2$ then $x = y_2$. Also $y_1 = 0$, and $S_t(c+1) = S_t(c) + x$. Therefore, for page allotment c, we are in step 1(d) of PRE3A and for page allotment c+1, we are in step 1(a) of PRE3A.

$\therefore$

$$U_{t+1}(c+1) = U_t(c+1) + x\ \cup\{y|y \in N_t(c+1),\ \mathrm{COUNT}(y,\ t+1,\ c+1) \geq T\}$$

$$= U_t(c) + x\ \cup\{y|y \in N_t(c),\ \mathrm{COUNT}(y,\ t+1,\ c+1) \geq T\}$$

Also since

$$\forall\, y \in N_t(c),\ \mathrm{COUNT}(y,\ t+1,\ c) = \mathrm{COUNT}(y,\ t,\ c) + 1$$

$$\text{and } \mathrm{COUNT}(y,\ t+1,\ c+1) = \mathrm{COUNT}(y,\ t,\ c+1) + 1,$$

we have,

$$U_{t+1}(c+1) = U_t(c) + x\ \cup\{y|y \in N_t(c),\ \mathrm{COUNT}(y,\ t+1,\ c) \geq T\}.$$

But from step 1(d) we have this

$$= U_{t+1}(c) + R_A(U_t(c), q, x).$$

Also,

$$S_{t+1}(c+1) = S_t(c+1) = S_t(c) + x$$

$$= S_{t+1}(c) + R_A(U_t(c), q, x).$$

From these, the required results follow.

<u>case (1b)</u>: $x \in U_t(c+1) \Rightarrow x \in U_t(c)$ or $x \notin U_t(c)$. Assume first that $x \in U_t(c)$ and assume that $U_t(c+1) = U_t(c) + y_1$. From step 1(b) of PRE3A,

$$U_{t+1}(c+1) = U_t(c+1) \cup \{y | y \in N_t(c+1), \text{COUNT}(y, t+1, c+1) \geq T\}$$

$$= U_t(c) + y_1 \cup \{y | y \in N_t(c), \text{COUNT}(y, t+1, c) \geq T\}$$

$$+ \text{if COUNT}(y_2, t+1, c+1) \geq T \text{ then } y_2$$

$$= U_{t+1}(c) + (y_1 \text{ or } y_2 \text{ or null})$$

Also,

$$S_{t+1}(c+1) = S_t(c+1) = S_t(c) + (y_1 \text{ or } y_2)$$

and $S_{t+1}(c) = S_t(c)$

$\therefore \quad S_{t+1}(c+1) = S_{t+1}(c) + y_1 \text{ or } y_2.$

This proves all the required things.

Next assume $x \notin U_t(c) \Rightarrow U_t(c+1) = U_t(c) + x.$

$$N_t(c+1) = N_t(c), \; S_t(c+1) = S_t(c) + x.$$

Therefore, for page allotment c+1, we are in step 1(b) and for page allotment c, we are in step 1(d) of PRE3A.

$$U_{t+1}(c+1) = U_t(c+1) \cup \{y | y \in N_t(c+1),\ \mathrm{COUNT}(y,\ t+1,\ c+1) \geq T\}$$

$$= U_t(c) + x \cup \{y | y \in N_t(c),\ \mathrm{COUNT}(y,\ t+1,\ c) \geq t\}$$

$$= U_{t+1}(c) + R_A(U_t(c),\ q,\ x).$$

Also,

$$S_{t+1}(c+1) = S_t(c+1) = S_t(c) + x,$$

$$S_{t+1}(c) = S_t(c) + x - R_A(U_t(c),\ q,\ x).$$

$$\therefore\ S_{t+1}(c+1) = S_{t+1}(c) + R_A(U_t(c),\ q,\ x).$$

From these the required results follow.

<u>case (1c)</u>: $x \notin \Gamma_t$ then we are in step 1(c) of PRE3A. We have

$$S_{t+1}(c+1) = U_{t+1}(c+1) = S_t(c+1) + x - R_A(U_t(c+1),\ q,\ x)$$

$$S_{t+1}(c) = U_t(c) = S_t(c) + x - R_A(U_t(c),\ q,\ x).$$

Let $U_t(c+1) = U_t(c) + y_1$.

Now since A is a stack algorithm,

$$R_A(U_t(c+1),\ q,\ x) = R_A(U_t(c),\ q,\ x) \text{ or } y_1.$$

Therefore,

$$S_{t+1}(c+1) = S_t(c+1) + x - (R_A(U_t(c),\ q,\ x) \text{ or } y_1)$$

$$= S_t(c) + y_1 + x - (R_A(U_t(c),\ q,\ x) \text{ or } y_1).$$

Therefore,

$$S_{t+1}(c+1) = S_{t+1}(c) + (R_A(U_t(c),\ q,\ x) \text{ or } y_1)$$

Also, since

$$N_{t+1}(c+1) = N_{t+1}(c) = \emptyset,$$

the required results follow.

case (1d): $x \notin S_t(c+1) \Rightarrow x \notin S_t(c)$. Therefore, we are in step 1(d) of PRE3A.

$$U_{t+1}(c+1) = U_t(c+1) + x - R_A(U_t(c+1), q, x)$$
$$\cup \{y | y \in N_t(c+1), \text{COUNT}(y, t+1, c+1) \geq T\}$$
$$= U_t(c) + y_1 + x - (R_A(U_t(c), q, x) \text{ or } y_1)$$
$$\cup \{y | y \in N_t(c), \text{COUNT}(y, t+1, c) \geq T\}$$
$$+ \text{ if COUNT}(y_2, t+1, c+1) \geq T \text{ then } y_2.$$

Also,

$$U_{t+1}(c) = U_t(c) + x - R_A(U_t(c), q, x)$$
$$\cup \{y | y \in N_t(c), \text{COUNT}(y, t+1, c) \geq T\}.$$

Since y_1 and y_2 cannot both be nonzero, we have,

$$U_{t+1}(c+1) = U_{t+1}(c) + (y_1 \text{ or } y_2 \text{ or } R_A(U_t(c), q, x))$$

Also,

$$S_{t+1}(c+1) = S_t(c+1) + x - R_A(U_t(c+1), q, x)$$
$$= S_t(c) + y_3 + x - (R_A(U_t(c), q, x) \text{ or } y_1)$$
$$= S_{t+1}(c) + (y_1 \text{ or } y_2 \text{ or } R_A(U_t(c), q, x)).$$

From these, the required result follows.

case (2): $r_{t+1} = \text{PRE}(x)$

Since ω satisfies the property P, we have, $x \notin \Gamma_t$ (or $x \notin S_t(c)$ for any c.). Assume $S_t(c+1) = S_t(c) + y_3$ and y_3 is not null.

(i) If $|S_t(c+1)| < c+1$, then $|S_t(c)| < c$.

Then, $N_{t+1}(c+1) = N_t(c+1) + x = N_t(c) + y_2 + x$

and $N_{t+1}(c) = N_t(c) + x$

Also since $y_2 = y_3$,

$N_{t+1}(c+1) = N_{t+1}(c) + y_3$

$S_{t+1}(c+1) = S_t(c+1) + x$

$= S_t(c) + y_3 + x$

and $S_{t+1}(c) = S_t(C) + x$

Therefore, $S_{t+1}(c+1) = S_{t+1}(c) + y_3$.

This proves the required result.

(ii) If $|S_t(c+1)| = c+1$, then $|S_t(c)| = c$.

Then, $S_{t+1}(c) = S_t(c)$, and $S_{t+1}(c+1) = S_t(c+1)$, etc.

Therefore, all the required results follow.

Now assume that y_3 is null, i.e. $S_t(c+1) = S_t(c)$.

(i) $|S_t(c)| < c$ which implies $|S_t(c+1)| < c+1$, then the required results are trivial to prove.

(ii) $|S_t(c)| = c$ which implies $|S_t(c+1)| = c < c+1$.

Therefore,

$S_{t+1}(c+1) = S_t(c+1) + x = S_t(c) + x$

and $S_{t+1}(c) = S_t(c)$ which implies $S_{t+1}(c+1) = S_{t+1}(c) + x$.

Also $N_{t+1}(c+1) = N_t(c+1) + x = N_t(c) + x$

and $N_{t+1}(c) = N_t(c)$ which implies $N_{t+1}(c+1) = N_t(c+1) + x$.

This proves the required result.

Since we have completed all possible cases of r_{t+1}, the proof is complete.

APPENDIX C

PL/I PROGRAMS FOR SEVERAL MATRIX ALGORITHMS

C.1 Cholesky Decomposition

```
CDSRFP:PROC(A,NSMALL,MSMALL);
/*THIS PROC FACTORS THE SYM.POS.DEF.MATRIX*/
/*A (OF ORDER NSMALL) INTO THE PROD. OF A */
/*LOWER TRIANG.& AN UPP.TRI.MAT.          */
/*IT IS A SUBMATRIX ALG.WITH LOOP REVERSAL*/
/*& FREEING & PREPAGING                   */
 N=NSMALL/MSMALL;
 DO I=1 TO N;
  /* FREE A<1,1>  */
  DO J=1 TO I-1;
   CALL FREE(A,I-1,J);
  END;

  /* PREPAGE SUBARRAYS C & D DIFFERENTIALLY   */
  DO K=I TO N;
   CALL PREPAGE(A,K,I);
  END;
  /* CARRY OUT D=D-R*R'     */
  DO J=1 TO I-1;
   DO I1=1 TO MSMALL;
    DO J1=1 TO I1;
     S=0;
     DO K1=1 TO MSMALL;
      S=S+A((I-1)*MSMALL+I1,(J-1)*MSMALL+J1)
         *A((I-1)*MSMALL+I1,(J-1)*MSMALL+J1);
     END;
     A((I-1)*MSMALL+I1,(I-1)*MSMALL+I1)=
     A((I-1)*MSMALL+I1,(I-1)*MSMALL+I1)-S;
    END;
   END;
  END;
  /*DO CHOLDEC OF THE DIAG.SUBMATRIX D */
  CALL CD(A((I-1)*MSMALL+1:I*MSMALL,(I-1)*MSMALL+1:I*MSMALL)
          ,MSMALL);
  /*INVERT SLT PART OF D & STORE INTO SUT PART OF D*/
  DO K=2 TO MSMALL;
   S1=A((I-1)*MSMALL+K,(I-1)*MSMALL+K);
   DO I1=1 TO K-1;
    S=A((I-1)*MSMALL+K,(I-1)*MSMALL+I1)/
      A((I-1)*MSMALL+I1,(I-1)*MSMALL+I1);
    DO J=I1+1 TO K-1;
     S=S+A((I-1)*MSMALL+K,(I-1)*MSMALL+J)*
        A((I-1)*MSMALL+I1,(I-1)*MSMALL+J);
    END;
    A((I-1)*MSMALL+I1,(I-1)*MSMALL+K)=S/S1;
   END;
  END;
  /*COMPUTE C=C-M*R' */
  /*REVERSE FOLLOWING LOOP WITH RE.TO I */
  IF MOD(I,2)=1 THEN
```

```
    DO;JLL=I+1;JHL=N;JSTEP=1;END;ELSE
    DO;JLL=N;JHL=I+1;JSTEP=-1;END;
   DO J=JLL TO JHL BY JSTEP;
    /*REVERSE FOLLOW.LOOP WITH RES.TO J */
    IF MOD(J,2)=1 THEN
     DO;KLL=1;KHL=I-1;KSTEP=1;END;ELSE
     DO;KLL=I-1;KHL=1;KSTEP=-1;END;
    DO K=KLL TO KHL BY KSTEP;
     DO J1=1 TO MSMALL;
      DO I1=1 TO MSMALL;
       S=0;
       DO K1=1 TO MSMALL;
        S=S+A((J-1)*MSMALL+J1,(K-1)*MSMALL+K1)*
            A((I-1)*MSMALL+I1,(K-1)*MSMALL+K1);
       END;
       A((J-1)*MSMALL+J1,(I-1)*MSMALL+I1)=
       A((J-1)*MSMALL+J1,(I-1)*MSMALL+I1)-S;
      END;
     END;
    END;
   END;
   /* MULT G<J,I> BY G<I,I> INVERSE */
   DO J=I+1 TO N;
    DCL S(MSMALL) FLOAT; /*TEMP. VECTOR */
    DO J1=1 TO MSMALL;
     DO I1=1 TO MSMALL;
      S(I1)=A((J-1)*MSMALL+J1,(I-1)*MSMALL+I1);
     END;
     DO I1=1 TO MSMALL;
      S=S(I1)/A((I-1)*MSMALL+I1,(I-1)*MSMALL+I1);
      DO K1=1 TO I1-1;
       S=S+S(K1)*A((I-1)*MSMALL+K1,(I-1)*MSMALL+I1);
      END;
      A((J-1)*MSMALL+J1,(I-1)*MSMALL+I1)=S;
     END;
    END;
   END;
END CDSRFP;
```

C.2 LU Decomposition

```
LUSRFP:PROC(A,NSMALL,MSMALL);
/*COMPUTES LU DECOMPOSITION OF MATRIX A OF ORDER*/
/*NSMALL & STORES L & U ON A;EMPLOYS A SUBMATRIX*/
/*ALGORITHM WITH LOOP REV.&FREEING&PREPAGING    */
 N=NSMALL/MSMALL;
 DO K=1 TO N;
  /* FREE SUBARRAY A(1,1) */
  DO I=1 TO K-1;
   CALL FREE(A,I,K-1);CALL FREE(A,K-1,I);
  END;
  /* PREPAGE SUBARRAYS Y & Z */
  DO I=K TO N;
   CALL PREPAGE(A,I,K);
  END;
  DO J=K+1 TO N;
   CALL PREPAGE(A,K,J);
  END;
  /* FORM D=D-R * X       */
  DO KK=1 TO K-1;
   DO I1=1 TO MSMALL;
    DO J1=1 TO MSMALL;
     S=0;
     DO KK1=1 TO MSMALL;
      S=S+A((K-1)*MSMALL+I1,(KK-1)*MSMALL+KK1)
         *A((KK-1)*MSMALL+KK1,(K-1)*MSMALL+J1);
     END;
     A((K-1)*MSMALL+I1,(K-1)*MSMALL+J1)=
     A((K-1)*MSMALL+I1,(K-1)*MSMALL+J1)-S;
    END;
   END;
  END;
  /* DO LU DECOMP OF D */
  CALL LU(A((K-1)*MSMALL+1:K*MSMALL,(K-1)*MSMALL+1:
            K*MSMALL),MSMALL);
  /*OCOMP. INV(LD), INV(UD) & STORE IN PAGE (N+1,1)  */
  DO K1=1 TO MSMALL;
   S1,A(NSMALL+K1,K1)=1/A((K-1)*MSMALL+K1,
                          (K-1)*MSMALL+K1);
   DO J=1 TO K1-1;
    S=0;
    DO I=J TO K1-1;
     S=S+A((K-1)*MSMALL+K1,(K-1)*MSMALL+I)*
        A(NSMALL+I,J);
    END;
    A(NSMALL+K1,J)=-S*S1;
   END;
   DO J=K1+1 TO MSMALL;
    S=A((K-1)*MSMALL+K1,(K-1)*MSMALL+J);
    DO I=K1+1 TO J-1;
     S=S+A((K-1)*MSMALL+K1,(K-1)*MSMALL+I)*
```

```
        A(NSMALL+I,JJ);
   END;
   A(NSMALL+K1,J)=-S;
  END;
END;
/* COMP. P=P-Q * X       */
/*REVERSE THE FOLLW.LOOP WITH RE. TO K */
IF MOD(K,2)=1 THEN
 DO;ILL=K+1 ;IHL=N;ISTEP=1;END;ELSE
 DO;ILL=N;IHL=K+1;ISTEP=-1;END;
DO I=ILL TO IHL BY ISTEP;
 /*REV. THE FOLLOW.LOOP WITH RE. I      */
 IF MOD(I,2)=1 THEN
  DO;KKL=1;KKH=K-1;KKSTEP=1;END;ELSE
  DO;KKL=K-1;KKH=1;KKSTEP=-1;END;
 DO KK=KKL TO KKH BY KKSTEP;
  DO I1=1 TO MSMALL;
   DO J1=1 TO MSMALL;
    S=0;
    DO KK1=1 TO MSMALL;
     S=S+A((I-1)*MSMALL+I1,(KK-1)*MSMALL+KK1)*
          A((KK-1)*MSMALL+KK1,(K-1)*MSMALL+J1);
    END;
    A((I-1)*MSMALL+I1,(K-1)*MSMALL+J1)=
    A((I-1)*MSMALL+I1,(K-1)*MSMALL+J1)-S;
   END;
  END;
 END;
END;
/* COMP. P=P*(INV(UU)+I)      */
DCL S(MSMALL) FLOAT /*TEMP. STORAGE VECTOR */;
DO I=K*MSMALL+1 TO NSMALL;
 DO J=1 TO MSMALL;
  S(J)=A(I,(K-1)*MSMALL+J);
 END;
 DO J=1 TO MSMALL;
  S=0;
  DO K1=1 TO J-1;
   S=S+S(K1)*A(NSMALL+K1,J);
  END;
  A(I,(K-1)*MSMALL+J)=A(I,(K-1)*MSMALL+J)+S;
 END;
END;
/*COMP. Z=Z-R*B          */
/*REVERSE THE FOLLOW.LOOP WITH RE.TO K*/
IF MOD(K,2)=1 THEN
   DO;JLL=K+1;JHL=N;JSTEP=1;END;ELSE
   DO;JLL=N;JHL=K+1;JSTEP=-1;END;
DO J=JLL TO JHL BY JSTEP;
 /*REV.THE FOLLOW.LOOP WITH RE. TO J  */
```

```
   IF MOD(J,2)=1 THEN
    DO;KKL=1;KKH=K-1;KKSTEP=1;END;ELSE
    DO;KKL=K-1;KKH=1;KKSTEP=-1;END;
   DO KK=KKL TO KKH BY KKSTEP;
    DO I1=1 TO MSMALL;
     DO J1=1 TO MSMALL;
      S=0;
      DO KK1=1 TO MSMALL;
       S=S+A((K-1)*MSMALL+I1,(KK-1)*MSMALL+KK1)*
            A((KK-1)*MSMALL+KK1,(J-1)*MSMALL+J1);
      END;
      A((K-1)*MSMALL+I1,(J-1)*MSMALL+J1)=
      A((K-1)*MSMALL+I1,(J-1)*MSMALL+J1)-S;
     END;
    END;
   END;
  END;
  /* COMPUTE Z=INV(LD)*Z */
  DO J=K*MSMALL+1 TO NSMALL;
   DO K1=1 TO MSMALL;
    S(K1)=A((K-1)*MSMALL+K1,J);
   END;
   DO I1=1 TO MSMALL;
    S=0;
    DO K1=1 TO I1;
     S=S+A(NSMALL+I1,K1)*S(K1);
    END;
    A((K-1)*MSMALL+I1,J)=S;
   END;
  END;
END LUSREP;
```

C.3 Gaussian Elimination

```
GOSSRFP:PROC(A,NSMALL,MSMALL);
/*THIS PROC DOES GAUSSIAN ELIM.ON MATRIX A(OF ORDER */
/*NSMALL)USING SUBMATRIX ALG.WITH LOOP REVERSAL &   */
/*ALSO USING FREEING & PREPAGING                    */
 N=NSMALL/MSMALL;
 /* PRELOAD AS MANY PAGES OF A AS POSSIBLE */
 /*DESIRABLE ONLY WHEN PAGE ALLOT.>2*N     */
 DO J=1 TO N;
  DO I=1 TO N;
   CALL PREPAGE(A,I,J);
 END;END;
 DO K=1 TO N;
 /*MAJOR LOOP OF THE ALGORITHM            */
  /* FREE  THE DEAD SUBARRAYS */
  DO I=1 TO K-1;
   CALL FREE(A,I,K-1);
   CALL FREE(A,K-1,I);
  END;
  /* DO GAUSSELIM OF D */
  CALL GOS(A((K-1)*MSMALL+1:K*MSMALL,(K-1)*
                   MSMALL+1:K*MSMALL),MSMALL);
  /*COMPLETE ELIM. ON SUBARRAYS R & C   */
  DO J=1 TO MSMALL;
   /*COMPUTE C<2>=C<2>/D<2,2>           */
   /*REVERSE LOOP WUTH RE. TO J         */
   IF MOD(J,2)=1 THEN
    DO;ILL=MSMALL*K+1;IHL=NSMALL;ISTEP=1;END;ELSE
    DO;ILL=NSMALL;IHL=K*MSMALL+1;ISTEP=-1;END;
   DO I=ILL TO IHL BY ISTEP;
    A(I,(K-1)*MSMALL+J)=A(I,(K-1)*MSMALL+J)/
              A((K-1)*MSMALL+J,(K-1)*MSMALL+J);
   END;
   /* C<3>=C<3>-C<2>*D<1,3>     */
   /*REVERSE THE FOLOW.LOOP OPP. TO PREV.LOP*/
   IF MOD(J,2)=0 THEN
    DO;ILL=K+1;IHL=N;ISTEP=1;END;ELSE
    DO;ILL=N;IHL=K+1;ISTEP=-1;END;
   DO I=ILL TO IHL BY ISTEP;
    DO J1=J+1 TO MSMALL;
     DO I1=1 TO MSMALL;
      A((I-1)*MSMALL+I1,(K-1)*MSMALL+J1)=A((I-1)*
      MSMALL+I1,(K-1)*MSMALL+J1)-A((I-1)*MSMALL+I1,
      (K-1)*MSMALL+J)*A((K-1)*MSMALL+J,(K-1)*MSMALL
      +J1);
     END;
    END;
   END;
   /* R<3>=R<3>-D<3,2>*R<2>      */
   /* REV WITH RE. TO J          */
   IF MOD(J,2)=1 THEN
```

```
      DO;JJL=K+1;JJH=N;JJSTEP=1;END;ELSE
      DO;JJL=N;JJH=K+1;JJSTEP=-1;END;
     DO JJ=JJL TO JJH BY JJSTEP;
      DO I1=J+1 TO MSMALL;
       DO JJ1=1 TO MSMALL;
        A((K-1)*MSMALL+I1,(JJ-1)*MSMALL+JJ1)=
        A((K-1)*MSMALL+I1,(JJ-1)*MSMALL+JJ1)-
        A((K-1)*MSMALL+I1,(K-1)*MSMALL+J)   *
        A((K-1)*MSMALL+J,(JJ-1)*MSMALL+JJ1);
       END;
      END;
     END;
    END;
    /* M=M-C*R                          */
    /*REV WITH RE. TO K                 */
    IF MOD(K,2)=1 THEN
     DO;ILL=K+1;IHL=N;ISTEP=1;END;ELSE
     DO;ILL=N;IHL=K+1;ISTEP=-1;END;
    DO I=ILL TO IHL BY ISTEP;
     /*REV WITH RE. TO I                */
     IF MOD(I,2)=1 THEN
      DO;JLL=K+1;JHL=N;JSTEP=1;END;ELSE
      DO;JLL=N;JHL=K+1;JSTEP=-1;END;
     DO J=JLL TO JHL BY JSTEP;
      DO I1=1 TO MSMALL;
       DO J1=1 TO MSMALL;
        S=0;
        DO K1=1 TO MSMALL;
         S=S+A((I-1)*MSMALL+I1,(K-1)*MSMALL+K1)*
              A((K-1)*MSMALL+K1,(J-1)*MSMALL+J1);
        END;
        A((I-1)*MSMALL+I1,(J-1)*MSMALL+J1)=
        A((I-1)*MSMALL+I1,(J-1)*MSMALL+J1)-S;
END GOSSRFP;
```

C.4 Gram-Schmidt Orthogonalization

```
ORTHSRFP:PROC(A,R,NSMALL1,ASMALL2,MSMALL);
/*THIS PROC FACTORS MATRIX A(OF SIZE NSMALL1 X NSMALL2*/
/*) INTO V * R, V HAS ORTHONORMAL COLUMNS & R IS UPPER*/
/* TRIAGULAR                                          */
 N1=NSMALL1/MSMALL;N2=NSMALL2/MSMALL;
 /*PREPAGE        ARRAY A,ADVISABLE IF PAGE ALLOT.>2*N1*/
  DO J=1 TO N2;
   DO I=1 TO N1;
    CALL PREPAGE(A,I,J);
  END;END;
 DO K=1 TO N2;
  /* FREE */
  DO I=1 TO N1 WHILE(K>1);
   CALL FREE(A,I,K-1);
  END;
  DO J=K-1 TO N2 WHILE(K>1);
   CALL FREE(R,K-1,J);
  END;
  /* PREPAGE */
  DO J=K TO N2;
   CALL PREPAGE(R,K,J);
  END;
  /* ORTHO. OF M VECTORS A(*,J) FOR K*M>=J>(K-1)*M */
  DO K1 =1 TO MSMALL;
   /* COMP.(K-1)*M+K1 TH. COL. OF R  */
   /*REV.LOOP WITH RE.TO     K1      */
   IF MOD(K1,2)=1 THEN
    DO;ILL=1;IHL=N1;ISTEP=1;END;ELSE
    DO;ILL=N1;IHL=1;ISTEP=-1;END;
   DO I=ILL TO IHL BY ISTEP;
    /*REV.LOOP WITH RE. TO    I      */
    IF MOD(I,2)=1 THEN
     DO;JLL=1;JHL=K1-1;JSTEP=1;END;ELSE
     DO;JLL=K1-1;JHL=1;JSTEP=-1;END;
    DO J=JLL TO JHL BY JSTEP;
     S=0;
     DO I1=1 TO MSMALL;
      S=S+A((I-1)*MSMALL+I1,(K-1)*MSMALL+J)*
          A((I-1)*MSMALL+I1,(K-1)*MSMALL+J);
     END;
     R((K-1)*MSMALL+J,(K-1)*MSMALL+K1)=
     R((K-1)*MSMALL+J,(K-1)*MSMALL+K1)+S;
    END;
   END;
   /*COMP. (K-1)*M+K1 TH. COL OF A */
   /*REVERSE WITH RE. TO K1        */
   DO I=IHL TO ILL BY -ISTEP;
    DO I1=1 TO MSMALL;
     S=0;
    /*REVERSE WITH RE. TO I1       */
```

```
    IF MOD(I1,2)=1 THEN
     DO;J1LL=1;J1HL=K1-1;J1STEP=1;END;ELSE
     DO;J1LL=K1-1;J1HL=1;J1STEP=-1;END;
    DO J1=J1LL TO J1HL BY J1STEP;
     S=S+A((I-1)*MSMALL+I1,(K-1)*MSMALL+J1)*
         R((K-1)*MSMALL+J1,(K-1)*MSMALL+K1);
    END;
    A((I-1)*MSMALL+I1,(K-1)*MSMALL+K1)=
    A((I-1)*MSMALL+I1,(K-1)*MSMALL+K1)-S;
   END;
  END;
  /*NORMALIZE JUST COMPUTED COL OF A */
  S=0;
  DO I=1 TO NSMALL1;
   S=S+A(I,(K-1)*MSMALL+K1)*A(I,(K-1)*MSMALL+K1);
  END;
  S=SQRT(S);R((K-1)*MSMALL+K1,(K-1)*MSMALL+K1)=S;
  /*REVERSE LOOP PERMANENTLY     */
  DO I=NSMALL1 TO 1 BY -1;
   A(I,(K-1)*MSMALL+K1)=A(I,(K-1)*MSMALL+K1)/S;
  END;
END;
/*COMPUTE R<2,3>=A<2>' * A<3>   */
/*REV LOOP WITH RE. TO K        */
IF MOD(K,2)=1 THEN
 DO;ILL=1;IHL=N1;ISTEP=1;END;ELSE
 DO;ILL=N1;IHL=1;ISTEP=-1;END;
DO I=ILL TO IHL BY ISTEP;
 /*REV LOOP WITH RE. TO I       */
 IF MOD(I,2)=1 THEN
  DO;KKL=K+1;KKH=N2;KKSTEP=1;END;ELSE
  DO;KKL=N2;KKH=K+1;KKSTEP=-1;END;
 DO KK=KKL TO KKH BY KKSTEP;
  DO KK1=1 TO MSMALL;
   DO J1=1 TO MSMALL;
    S=0;
    DO I1=1 TO MSMALL;
     S=S+A((I-1)*MSMALL+I1,(K-1)*MSMALL+J1)*
         A((I-1)*MSMALL+I1,(KK-1)*MSMALL+KK1);
    END;
    R((K-1)*MSMALL+J1,(KK-1)*MSMALL+KK1)=S+
    R((K-1)*MSMALL+J1,(KK-1)*MSMALL+KK1);
   END;
  END;
 END;
END;
/*COMPUTE A<3>=A<3> - A<2> * R<2,3>     */
/*REV IN A DIRECTION OPP. TO PREV. LOOP */
DO I=IHL TO ILL BY -ISTEP;
 /*REV WITH RE. TO I    */
```

```
    IF MOD(I,2)=1 THEN
     DO;JLL=K+1;JHL=N2;JSTEP=1;END;ELSE
     DO;JLL=N2;JHL=K+1;JSTEP=-1;END;
    DO J=JLL TO JHL BY JSTEP;
     DO I1=1 TO MSMALL;
      DO J1=1 TO MSMALL;
       S=0;
       DO K1=1 TO MSMALL;
        S=S+A((I-1)*MSMALL+I1,(K-1)*MSMALL+K1)*
            R((K-1)*MSMALL+K1,(J-1)*MSMALL+J1);
       END;
       A((I-1)*MSMALL+I1,(J-1)*MSMALL+J1)=
       A((I-1)*MSMALL+I1,(J-1)*MSMALL+J1)-S;
END ORTHSREP;
```

C.5 Matrix Multiplication

```
MMSRFP:PROC(A,B,C,NSMALL,MSMALL);
/* THIS PROC MULTIPLIES MATRICES B & C & STORES */
/* INTO A, BY SUBMATRICES;USES REVERSAL,FREEING */
/* & PREPAGING                                  */
 N=NSMALL/MSMALL;
 /*PREPAGE AS MUCH OF ARRAY C AS POSSIBLE      */
 /*DESIRABLE IF PAGE ALLCT. > 2*N              */
 DO J=1 TO N;
  DO I=1 TO N;
   CALL PREPAGE(C,I,J);
 END;END;
 /*MAIN LOOP OF ALG.    */
 DO I=1 TO N;
  /*FREE DEAD PORTIONS OF A & B    */
  DO I1=1 TO N WHILE(I>1);
   CALL FREE(A,I-1,I1);CALL FREE(B,I-1,I1);
  END;
  /*PREPAGE PORTIONS OF A & B      */
  DO I1=1 TO N;
   CALL PREPAGE(B,I,I1);
   CALL PREPAGE(A,I,I1);
  END;
  /*REV LOOP WITH RE. TO I         */
  IF MOD(I,2)=1 THEN
   DO;JLL=1;JHL=N;JSTEP=1;END; ELSE
   DO;JLL=N;JHL=1;JSTEP=-1;END;
  DO J=JLL TO JHL BY JSTEP;
   /*REV LOOP WITH RE  TO J        */
   IF MOD(J,2)=1 THEN
    DO;KLL=1;KHL=N;KSTEP=1;END; ELSE
    DO;KLL=N;KHL=1;KSTEP=-1;END;
   DO K=KLL TO KHL BY KSTEP;
    DO I1=1 TO MSMALL;
     DO J1=1 TO MSMALL;
      S=0;
      DO K1=1 TO MSMALL;
       S=S+B((I-1)*MSMALL+I1,(K-1)*MSMALL+K1)*
           C((K-1)*MSMALL+K1,(J-1)*MSMALL+J1);
      END;
      A((I-1)*MSMALL+I1,(J-1)*MSMALL+J1)=S+
      A((I-1)*MSMALL+I1,(J-1)*MSMALL+J1);
END MMSRFP;
```

C.6 CDS

```
CDS:PROC(A,NSMALL,MSMALL);
/*THIS PROC FACTORS SYM.POS.DEF. MATRIX A(OF ORDER*/
/*NSMALL) INTO G*G' ,G IS LOWER TIANG.THIS IS A   */
/*SUBMATRIX ALGORITHM                             */
 N=NSMALL/MSMALL;
 DO I=1 TO N;
  /* COMPUTE D=D-R*R'  */
  DO J=1 TO I-1;
   DO II=1 TO MSMALL;
    DO J1=1 TO I1;
     S=0;
     DO K1=1 TO MSMALL;
      S=S+A((I-1)*MSMALL+I1,(J-1)*MSMALL+K1)*
          A((I-1)*MSMALL+J1,(J-1)*MSMALL+K1);
     END;
     A((I-1)*MSMALL+I1,(I-1)*MSMALL+J1)=
     A((I-1)*MSMALL+I1,(I-1)*MSMALL+J1)-S;
    END;
   END;
  END;
  /* DO DECOMPOSITION OF DIAG.SUBM. D       */
  CALL CD(A((I-1)*MSMALL+1:I*MSMALL,
            (I-1)*MSMALL+1:I*MSMALL),MSMALL);
  /*INVERT SLT PART OF D & STORE IN SUT PART OF D*/
  DO K=2 TO MSMALL;
   S1=A((I-1)*MSMALL+K,(I-1)*MSMALL+K);
   DO I1=1 TO K-1;
    S=A((I-1)*MSMALL+K,(I-1)*MSMALL+I1)/
      A((I-1)*MSMALL+I1,(I-1)*MSMALL+I1);
    DO J=I1+1 TO K-1;
     S=S+A((I-1)*MSMALL+K,(I-1)*MSMALL+J)*
         A((I-1)*MSMALL+I1,(I-1)*MSMALL+J);
    END;
    A((I-1)*MSMALL+I1,(I-1)*MSMALL+K)=S/S1;
   END;
  END;
  /*COMPUTE C=C-M*R'  */
  DO J=I+1 TO N;
   DO K=1 TO I-1;
    DO J1=1 TO MSMALL;
     DO I1=1 TO MSMALL;
      S=0;
      DO K1=1 TO MSMALL;
       S=S+A((J-1)*MSMALL+J1,(K-1)*MSMALL+K1)*
           A((I-1)*MSMALL+I1,(K-1)*MSMALL+K1);
      END;
      A((J-1)*MSMALL+J1,(I-1)*MSMALL+I1)=
      A((J-1)*MSMALL+J1,(I-1)*MSMALL+I1)-S;
     END;
    END;
```

```
    END;
   END;
   /* MULT G<J,I> BY G<I,I> INVERSE */
   DCL S(MSMALL) FLOAT; /*TEMP.VECTOR */
   DO J=I+1 TO N;
    DO J1=1 TO MSMALL;
     DO I1=1 TO MSMALL;
      S(I1)=A((J-1)*MSMALL+J1,(I-1)*MSMALL+I1);
     END;
     DO I1=1 TO MSMALL;
      S=S(I1)/A((I-1)*MSMALL+I1,(I-1)*MSMALL+I1);
      DO K1=1 TO I1-1;
       S=S+S(K1)*A((I-1)*MSMALL+K1,(I-1)*MSMALL+I1);
      END;
      A((J-1)*MSMALL+J1,(I-1)*MSMALL+I1)=S;
END CDS;
```

APPENDIX D

ALGORITHM AP

ALG-AP:

/* Constructs the USE-LIST and uses it to insert instructions in an OL/2 program */

Step AP-1 [Data Structure]:

[A]: A node in the USE-LIST is one of the seven types: ARRAY, SUBARRAY, FOR-BEGIN, FOR-END, DEAD-SUBARRAY, END-OF-LIFETIME, END-OF-SCOPE.

(1) A node of type ARRAY has three fields: NAME, LP and NEXT. NAME is a pointer to the ACB of the array, LP is the lifetime pointer of the array, and NEXT is the static link in the USE-LIST.

(2) A node of type SUBARRAY has the following fields: NAME, X, KC, NEXT, LP.
NAME is a pointer to the ACB of the subarray, X consists of four subfields: r_{I-1}, r_I, c_{J-1}, c_J, each indicating a boundary expression of the subarray. NEXT and LP are as in (1) above. KC is a (possibly empty) set of subfields, each threaded on a dynamic link (k-chain) of some FOR loop (and indicating that this subarray is dynamic with respect to that loop).

(3) A node of type FOR-BEGIN has three fields: NAME, KC and NEXT. NAME contains some unique identifier of the FOR loop; we have assumed that the loop control variable is such an identifier. NEXT is as in (1) above. The field KC originates the k-chain of the loop.

(4) A node of type FOR-END has three fields: BP, NEXT and KC. BP is a back-pointer to the associated FOR-BEGIN node, NEXT is as in (1) above, KC is the end of the k-chain of the loop.

(5) A node of type DEAD-SUBARRAY has three fields: NAME, X and KC. All these three fields are as in (2) above.

(6) A node of type END-OF-LIFETIME has two fields: RP and NEXT. RP is a return-pointer to the ARRAY or SUBARRAY node of this (sub)array. NEXT is as in (1) above.

(7) A node of type END-OF-SCOPE has two fields: NAME and NEXT. NAME is the name of the array and NEXT is as before.

[B]: The ACB of each array is modified to have two additional pointer fields, IP and CP.

[C]: Definitions: (Scope and lifetime)

(1) S(A) = if A is of type ARRAY then ($\min_{B \subseteq A}$ IP[B], $\max_{B \subseteq A}$ LP[B]);

else if A is of type FOR-BEGIN then

(pointer to FOR-BEGIN[A], pointer to FOR-END[A]);

(2) L(A) = if A is of type ARRAY or SUBARRAY then (IP[A], LP[A]);

Step AP-2 [Build up the USE-LIST during the syntax analysis phase]:

[A] if SCAN = 'FOR' then

do;

Create a node of type FOR-BEGIN, pointed to by P, and link it to the USE-LIST (on the static link); set NAME[P] = 'k';

KC[P] = null;

end;

[B] if SCAN = 'A' /* an array */ then if CP[A] = null then
/* the first occurrence of A */
do;
Create a node of type ARRAY, pointed to by P, and link it to
the USE-LIST; set IP[A] = P; CP[A] = P; NAME[P] = A; LP[P] = null;
end; else /* not the first occurrence */ CP[A] = P;

[C] if SCAN = 'B' then /* a subarray, B = A<I,J> */
if CP[B] = null then /* first occurrence */
do;
Create a node of type SUBARRAY, pointed to by P, and link it
to the USE-LIST; set IP[B] = CP[B] = P; NAME[P] = B; LP[P]=null;
X[P] = (r_{I-1}, r_I, c_{J-1}, c_J); KC[P] = null;
end; else /* not first occurrence */ CP[B] = P;

[D] if SCAN = 'END' then /* end of a FOR loop */
do;
Create a node of type FOR-END, pointed to by P, and link it
to the USE-LIST; link this node to the k-chain of the corresponding
FOR loop and set BP[P] = pointer to the matching FOR-BEGIN node;
end;

Step AP-3 [This step modifies the USE-LIST to add dead subarrays, lifetime pointer, correct lifetimes and end of scope nodes. This is done in a post-syntax-analysis, precoding phase]:

[A] [fill up lifetime pointers and create end-of-lifetime nodes]:

Scan the USE-LIST sequentially.

if SCAN = ARRAY[P] or SUBARRAY[P] then

do;

B = NAME [P]; LP[P] = CP[B]; Create a node of type END-OF-LIFETIME, pointed to by Q, and insert in the USE-LIST at LP[P]; set RP[Q] = P;

end;

[B] [link dynamic subarrays on the k-chain]:

Scan the USE-LIST sequentially.

if the current nest level l > 0 then if SCAN = SUBARRAY[P] then

do;

B = NAME [P];

for all 'FOR' loops f do;

if $S(f) \cap L(B) \neq \varphi$ then

do;

k = NAME[f];

if P(B, k) = T then link B to the k-chain of loop f;

end;

end; end;

[C] [modify the USE-LIST to take care of the overlapping lifetimes, introduce end-of-scope node, enter dead-subarrays]:

(1) [modify overlapping lifetimes]:

for all arrays A such that $S(A) \neq \varphi$, do;

/* let l be the level of a subarray B of A. Here the level means, the level in ACB tree structure */

```
l_min = min l(B);   l_max = max l(B);
        B⊆A                 B⊆A

     /* note that l(A) = 1,  l(A<1, 2>) = 2  */

for  j = l_max  to l_min - 1,  do;

  for all D⊆A  such that  l(D) = j  do;

    let Q be the pointer to the node D in the USE-LIST; let
    the lowest level ancestor of D in the USE-LIST be denoted
    by E and P points to its node in the USE-LIST;
    if (IP[E] > IP[D] ) and ( LP[P] ≥ LP[Q] )  then
      do;
        CP[D] =  LP[Q] = LP[P];  delete the END-OF-LIFETIME
        node of D;
      end;
    if ( LP[P] < LP[Q] ) then
      do;
        CP[E] = LP[P] = LP[Q]; delete the END-OF-LIFETIME
        node of E; RP[Q] = P;
        if  (IP[E] ≤ IP[D] ) then
          do;
            delete node D from the USE-LIST; CP[D] = IP[D] = null;
end [C] (1);
```

(2) [insert the END-OF-SCOPE nodes]:

for all arrays A such that there is a subarray B of A such that B ϵ USE-LIST do;

set (P,Q) = S(A); Create a node of type END-OF-SCOPE and insert it in the USE-LIST at Q; NAME[Q] = A;

end;

(3) [create dead subarrays and link to USE-LIST]:

for all arrays $\notin$ USE-LIST such that there is a subarray B of A such that B ϵ USE-LIST do;

j=max $\{$l(B) | B$\subset$A, B ϵ USE-LIST, there is a k such that P(B,k) = T $\}$;

construct a sequence of subarrays of A at level j;

name the above sequence DEAD-LIST(A);

if (B ϵ DEAD-LIST(A)) and ((B ϵ USE-LIST) or (for all k P(B,k)=F)) then delete B from DEAD-LIST(A);

for each FOR loop f do;

for each B ϵ DEAD-LIST(A) do;

let the loop control variable of loop f be k;

let Q point to FOR-BEGIN node of f;

if S(f) $\cap$ S(A) $\neq \varphi$ then if P(B,k) = T then do;

create a node of type DEAD-SUBARRAY, pointed to by P, and link it to the k-chain of f;

NAME[P] = B; X[P] = $(r_{I-1}, r_I, c_{J-1}, c_J)$;

end;

end Step AP-3;

Step AP-4 [use of the USE-LIST in the coding phase to do freeing and prepaging]:

[A] [freeing]:

if at lexical level $\ell = 0$ then on scanning an END-OF-LIFETIME node of B do;

if B is an array then generate instruction: 'FREE(B)';

if B is a subarray then generate instructions:

'IF for all x ϵ X, MOD(x,m)=0 THEN FREE(B)';

end;

on scanning an END-OF-SCOPE node of A generate the instruction:

'FREE(A)';

if at level $\ell > 0$ then on scanning an END-OF-LIFETIME or END-OF-SCOPE node of B do the same as above except that the code is inserted after the level ℓ is reduced to zero, i.e. after exit from the outermost loop;

[B] [prepaging outside the loop]:

if at level $\ell = 0$ then generate instructions to prepage a few nodes in advance;

if at level $\ell > 0$ then prior to generating code for the loop, generate instructions to prepage all the (sub)arrays in the USE-LIST from the FOR-BEGIN node to the FOR-END node along the static link;

[C] [freeing within a loop]:

(level $\ell > 0$)

for all dead-subarrays B along the dynamic chain of the loop, generate the following code just after the backward branch target of the loop:

'IF ((for all x ϵ X_1 , $f_{k_{i+1}}(x) \leq 0$) AND

(for all x ϵ X_2 , $f_{k_{i+1}}(x) \geq 0$)) THEN

IF for all $x \in X$,MOD($x(k_{i+1})$,m) = 0 THEN FREE(B)' ;

[D] [incremental prepaging within a loop]:

(level $\ell > 0$)

for all dynamic subarrays B along the dynamic chain of the loop, insert the following code after the backward branch target of the loop:

'IF ((there exists $x \in X_1$ such that $f_{k_{i+1}}(x) = -1$) OR

(there exists $x \in X_2$ such that $f_{k_{i+1}}(x) = 1$)) THEN

IF there exists $x \in X_0$ such that MOD($x(k_i)$,m) = 0

THEN PREPAGE(B) ' ;

end ALG-AP;

VITA

Kishor Shridharbhai Trivedi was born in Bhavnagar, Gujarat State (India), on August 20, 1946. He received the Bachelor's degree in Electrical Engineering from the Indian Institute of Technology, Bombay (India) in 1968. Between 1968 and 1970, he was first employed by Larsen and Toubro India Limited as an Apprentice Engineer and then by International Business Machines (WTC) of India as a Customer Engineer.

He joined the University of Illinois, Urbana, as a graduate research assistant in the Department of Computer Science in 1970. He conducted research in the area of Digital Computer Arithmetic under the guidance of Professor James E. Robertson. He obtained his Master's degree in Computer Science in 1972. Since then, he conducted research in the area of Operating Systems and Numerical Analysis under the guidance of Professor J. Richard Phillips. He is the coauthor with J. E. Robertson of a paper entitled, "The Status of Investigations into Computer Hardware Design Based on the Use of Continued Fractions," which was published in IEEE Transactions on Computers, June 1973.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-74-662	2.	3. Recipient's Accession No.

4. Title and Subtitle

PREPAGING AND APPLICATIONS TO STRUCTURED ARRAY PROBLEMS

5. Report Date: July 1974

6.

7. Author(s): Kishor Shridharbhai Trivedi

8. Performing Organization Rept. No. UIUCDCS-R-74-662

9. Performing Organization Name and Address

University of Illinois at Urbana-Champaign
Department of Computer Science
Urbana, Illinois 61801

10. Project/Task/Work Unit No.

11. Contract/Grant No.: US NSF GJ-328

12. Sponsoring Organization Name and Address

National Science Foundation
Washington, D. C.

13. Type of Report & Period Covered: Doctoral - 1974

14.

15. Supplementary Notes

16. Abstracts

A demand prepaging algorithm is defined and proved to be an optimal demand prepaging algorithm. A variable-memory prepaging algorithm PWS is also defined and is based on Denning's WS algorithm. It is shown that PWS incurs zero page faults. Both of these algorithms, however, cannot be used in practice since they require that the reference string be known in advance. Therefore, we describe several practical prepaging algorithms.

Data paging is of primary concern for problems with large data bases and for many types of array problems. In particular, we show that prepaging reduces the paging problems of array algorithms operating on large arrays. We also show that the use of a submatrix algorithm considerably improves the locality. Finally, we consider methods of automating these performance-improvement techniques by means of a compiler in the context of a structure-array-language.

17. Key Words and Document Analysis. 17a. Descriptors

Array Language
Compiler
Demand Paging
Locality
Matrix Algorithm
Multiprogramming
Paging Performance
Prepaging
Virtual Memory

b. Identifiers/Open-Ended Terms

c. COSATI Field/Group

18. Availability Statement: RELEASE UNLIMITED

19. Security Class (This Report): UNCLASSIFIED

20. Security Class (This Page): UNCLASSIFIED

21. No. of Pages

22. Price: 200